Cpo-models for second order lambda calculus with recursive types and subtyping

Poll, E.

Published: 01/01/1991

Document Version
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
Eindhoven University of Technology
Department of Mathematics and Computing Science

Cpo-models for second order lambda calculus
with recursive types and subtyping

by

Erik Poll

Computing Science Note 91/07
Eindhoven, September 1991
This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

All rights reserved
editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.
Cpo-models for second order lambda calculus with recursive types and subtyping

Erik Poll*

Abstract

In this paper we present constructions of cpo models for second order lambda calculi with recursive types and/or subtyping. The model constructions are based on a model construction by ten Eikelder and Hemerik for second order lambda calculus with recursive types ([tEH89a]). The models will be compatible with conventional denotational semantics.

For each of the systems we consider, the general structure of an environment model for that system is described first. For the systems with subtyping we prove coherence, i.e. that the meaning of a term is independent of which particular type derivation we consider. The actual model constructions are then based on a standard fixed-point result for ω-categories. The combination and interaction of recursive types and subtyping does not pose any problems.

*supported by the Dutch organization for scientific research (NWO).
### Contents

1 **Introduction** .................................................. 2

2 **Second order lambda calculus** .......................... 5
   2.1 Syntax ..................................................... 5
   2.2 Semantics: general model definition .................... 7
   2.3 The construction of a cpo model ......................... 11

3 **Recursive types** ............................................. 15
   3.1 \(\Lambda\mu_1\) ............................................... 16
      3.1.1 Syntax ................................................ 16
      3.1.2 Semantics: general model definition ............... 16
      3.1.3 The construction of a cpo model ................... 17
   3.2 \(\Lambda\mu_2\) ............................................... 18
      3.2.1 Syntax ................................................ 18
      3.2.2 Semantics: general model definition ............... 18
      3.2.3 The construction of a cpo model ................... 19
   3.3 \(\Lambda\mu_3\) ............................................... 20
      3.3.1 Syntax ................................................ 20
      3.3.2 Semantics: general model definition ............... 21
      3.3.3 The construction of a cpo model ................... 21

4 **Subtyping** .................................................. 25
   4.1 Syntax ..................................................... 25
   4.2 Semantics: general model definition .................... 26
   4.3 The construction of a cpo model ......................... 29

5 **Recursive types and subtyping** ...................... 34
   5.1 Syntax ..................................................... 34
   5.2 Semantics: general model definition .................... 34
   5.3 The construction of a cpo model for \(\Lambda \leq \mu_1\) ...... 35
   5.4 The construction of a cpo model for \(\Lambda \leq \mu_3\) ...... 38
   5.5 The construction of a cpo model for \(\Lambda \leq \mu_2\) ...... 40

6 **Conclusion** .................................................. 41

Appendix: Coherence ............................................... 44
1 Introduction

The second order lambda calculus (or polymorphic lambda calculus) was discovered independently by Girard [Gir72] and Reynolds [Rey74]. It is an extension of the simple typed lambda calculus: not only terms but also types can be passed as parameters. This means that besides abstraction over term variables and application of terms to terms we also have abstraction over type variables and application of terms to types.

In this paper we consider two extensions of the second order lambda calculus: subtyping and recursive types. We first construct a model for the second order lambda calculus, and then show how this construction can be adapted to include subtyping and recursive types.

Both subtyping and recursive types are interesting from the point of view of programming languages. Recursive types can be used to make types such as list and trees. Also fixed point operators, which cannot be typed in second order lambda calculus, can be typed using recursive types.

Subtyping can also be found in programming languages: in combination with labelled records it corresponds with inheritance in object-oriented languages. This form of subtyping can be found in Cardelli and Wegner's language Fun [CW85], and more recently also in Quest [CL90].

We only consider a very simple form of subtyping. We do not have labelled records or bounded quantification, as for instance in Fun, but instead all subtyping will be based on a subtype relation on a set of base types. For example, if we have base types $\text{int}$ and $\text{real}$ we could have $\text{int} \leq \text{real}$, i.e. $\text{int}$ is a subtype of $\text{real}$. In the final section we will show that the incorporation of bounded quantification and record types in the models is straightforward.

Several models for second order lambda calculus are known, for example models based on partial equivalence relations [Gir72], the closure model [Mac79], the finitary projection model [ABL86] and models based on qualitative domains [Gir86].

The model constructions in this paper are based on a model construction by ten Eikelder and Hemerik for second order lambda calculus with recursive types [tEH89a]. The models are more oriented towards programming language semantics, and are compatible with conventional denotational semantics. Types will be interpreted as cpos, which are commonly used as semantic domains in denotational semantics. Directed cpos or complete lattices could also be used. Recursion at term level can then be handled by the usual fixed point theory for cpos. Because types are interpreted as cpos we do not have empty types.

A pleasing aspect of the model constructions is that other type constructors, such as $\Sigma$ (existential types), $\times$ (Cartesian product), $+$ (separated sum), $\otimes$ (smashed product), $\oplus$ (coalesced sum) or $(\cdot)_\bot$ (lifting) can easily be added.

Coercion functions are used to give the semantics of subtyping: if a type $\sigma$ is a subtype of a type $\tau$, we have a coercion function from the cpo for $\sigma$ to the cpo for $\tau$.

The main problem in giving a model for systems with subtyping is that meanings are defined by induction on type derivations, and because of the subtyping many type derivations will be possible. We must prove that all derivations for a term give the same meaning, which is called coherence. Examples of coherence proofs can be found in [BTCGS89] and [CG90]. In both papers coercions are used to interpret a second order $\lambda$-calculus with subtyping, and coherence is proved for this interpretation.
Providing a semantics for systems which have both subtyping and recursive types has long been regarded as problematic. Models that incorporate subtyping based on partial equivalence relations, such as Bruce and Longo’s model for \textit{Fun} [BL88] and Cardelli and Longo’s model for (a part of) \textit{Quest} [CL90], cannot easily be extended to model recursive types. Using the method described in [BTCGS89] however, a semantics for subtyping and recursive types (but not for subtyping on recursive types) can be constructed using a semantics that models recursive types but does not model subtyping. For the models we construct the combination and interaction of recursive types and subtyping does not pose any problems. There will be no need to restrict the recursive types to those without negative occurrences of the type variables.

Before we combine subtyping and recursive types, we first consider them separately. We will consider several ways to define equality for recursive types, each resulting in slightly different systems.

For all resulting systems we give general model definitions similar to the definition of a Bruce-Meyer-Mitchell environment model [BMM90], and we construct cpo models based on those general model definitions. An advantage of the general model definitions is that we can prove properties not just for one particular model but for all models that fit the general model definition. For example, for the systems with subtyping we can prove that a model is coherent, if the coercions satisfy certain conditions.

Once we have the general model definition, the construction of a model is relatively simple.

For the systems without subtyping, the model constructions are slight modifications of the one given in [TEHS9a]. Constructing a model is a question of solving the set of recursive domain equations given by the general model definition. Because types are interpreted as cpos, the problem of the contravariance of \( \sigma \rightarrow \tau \) in \( \sigma \) can be overcome in the standard way, by working in a category of embedding-projection pairs, a technique described in [SP82][BH88]. A solution for the recursive domain equation is then found using a standard fixed point construction for \( \omega \)-continuous functors on a suitable product category of \( CPO_{PR} \) (an inverse-limit construction).

For the systems with subtyping, we not only have to solve the recursive domain equations, but we also have to find coercion functions between the domains of types that are in the subtype relation. For the semantics to be coherent, the coercions have to satisfy certain conditions. Together, the domains and coercions form a functor from a category corresponding with the subtype relation on types to \( CPO \). Such a functor, satisfying both the recursive domain equations and the coherence conditions, is again found by an inverse limit construction, only this time in a functor category. The problem of the contravariance of \( \rightarrow \) is overcome in the same way as for the systems without subtyping, viz. by using projection-embedding pairs. For the rather technical proofs of the categorical properties needed for this construction we refer to [Pol91].

In the following section we give a short description of the second order lambda calculus. The version we describe is identical to the one described in [BMM90],[Mit84] and [ABL86]. We then give the definition of a Bruce-Meyer-Mitchell environment model and construct a cpo model based on that definition.

In section 3 we consider several ways to extend second order lambda calculus with recursive types. For each possibility we give a general model definition and we construct a cpo model.

In section 4 we then describe the second order lambda calculus extended with subtyping, and again we give a general model definition and construct a cpo model, and in section 5 we consider the second order lambda calculus with both subtyping and recursive types.
Finally in the concluding section we indicate how bounded quantification, record types and other extensions can be included in the model, and we briefly discuss the model constructions.
2 Second order lambda calculus

2.1 Syntax

We will now give a short description of the second order lambda calculus \( \Lambda \) for short. The system described here is the same as in \cite{BMM90}, \cite{ABL86} and \cite{Mit84}.

We distinguish three sorts of expressions: kinds, constructors and terms. Every term has a type. Types are made using constructors. In fact, the types themselves are also constructors. The constructors also have "types", which we call kinds.

kinds

The set of kinds is given by

\[ \kappa ::= \ast | \kappa_1 \rightarrow \kappa_2 \]

Kinds are the "types" of construction expressions.

constructors and their kinds

Let \( C_{cons} \) be a set of constructor constants and \( V_{cons} \) be a set of constructor variables. All constructors constants have a specified kind, which we will write as a superscript when necessary. First we define the set of pseudo-constructors over \( C_{cons} \) and \( V_{cons} \), of which the set of constructor expressions will be a subset.

The set of pseudo-constructors over \( C_{cons} \) and \( V_{cons} \) is given by:

\[ \sigma = c | \alpha | \sigma_1 \sigma_2 | (\Lambda \alpha : \kappa.\sigma) \]

where \( c \in C_{cons} \), \( \alpha \in V_{cons} \) and \( \kappa \) a kind.

The system \( \Lambda \) we describe here is not quite the same as Girard’s system \( F \) or \( \lambda 2 \) in Barendregt’s cube \cite{Bar9}, because we allow abstraction over all kinds here and not just over types. In the terms however, we shall only allow abstraction over types.

Constructors are those pseudo-constructors for which a kind can be derived in a context. A context here is a syntactic kind assignment, i.e. a partial function from \( V_{cons} \) to the set of kinds. So a context assigns kinds to constructor variables. We write \( \Gamma \vDash \sigma : \kappa \) if we can derive that in context \( \Gamma \) the constructor \( \sigma \) has kind \( \kappa \), using the following rules:

\[ \begin{align*}
\Gamma \vdash e^\kappa : \kappa & \quad (e^\kappa \in C_{cons}) \\
\Gamma, \alpha : \kappa_1 \vdash \alpha : \kappa & \quad (\alpha \in V_{cons}) \\
\Gamma \vdash (\Lambda \alpha : \kappa_1.\sigma) : \kappa_1 \rightarrow \kappa_2 & \quad (\Rightarrow I) \\
\Gamma \vdash \sigma : \kappa_1 \Rightarrow \kappa_2 & \quad \Gamma \vdash \tau : \kappa_1 \Rightarrow \kappa_2 (\Rightarrow E)
\end{align*} \]

Constructor expressions of kind \( \ast \) will be called type expressions. \( \Gamma \vdash \sigma : \ast \) means that \( \sigma \) is a type in context \( \Gamma \).

We assume that \( C_{cons} \) contains the following constants:

\[ \begin{align*}
\rightarrow & : \ast \Rightarrow (\ast \Rightarrow \ast) \quad \text{ (for function types)} \\
\Pi & : (\ast \Rightarrow \ast) \Rightarrow \ast \quad \text{ (for polymorphic types)}
\end{align*} \]

We also have constructor constants of kind \( \ast \), which we call the the base types. For example, these might include the types \textit{bool}, \textit{int} or \textit{real}. 

So, for example

\[ \alpha : \star \vdash \alpha : \star \Rightarrow \star \]
\[ \alpha : \star \vdash \alpha \alpha : \star \]

\( \vdash \) will be written infix.

\[ \alpha : \star \vdash \alpha \to \alpha : \star \]
\[ \vdash (\lambda \alpha : \star. \alpha \to \alpha) : \star \Rightarrow \star \]
\[ \vdash \Pi(\lambda \alpha : \star. \alpha \to \alpha) : \star \]

The constructor expressions form a simple typed lambda calculus. Equality on constructor expressions is \( \beta \eta \)-equality. If in a context \( \Gamma \) constructors \( \sigma \) and \( \tau \) are equal, we write \( \Gamma \vdash \sigma =_c \tau \).

The following are well-known properties of simple typed \( \lambda \)-calculus.

1 property

(i) the kind of a constructor in a given context is unique

(ii) equal constructors have the same kind

\( \Box \)

terms and their types

We will now define the set of term expressions, in the same way as we defined the set of constructor expressions.

Let \( C_{\text{term}} \) be a set of term constants and \( V_{\text{term}} \) be a set of term variables. All term constants have a specified type, which we will write as a superscript when necessary.

We first define the set of pseudo-terms over \( C_{\text{term}} \) and \( V_{\text{term}} \), of which the set of term expressions will be a subset.

The set of pseudo-terms over \( C_{\text{term}} \) and \( V_{\text{term}} \) is given by:

\[ M = c \mid x \mid (\lambda x : \sigma.M) \mid M_1 M_2 \mid (\Lambda \alpha : \star.M) \mid M \sigma \]

where \( x \in V_{\text{term}} \), \( c \in C_{\text{term}} \), \( \alpha \in C_{\text{cons}} \) and \( \sigma \) a pseudo-constructor.

So we have abstraction over term variables, \( (\lambda x : \sigma.M) \), and we have abstraction over type variables, \( (\Lambda \alpha : \star.M) \), and the corresponding forms of application: of a term to a term, \( M_1 M_2 \), and of a term to a type, \( M \sigma \).

Terms are those pseudo-terms for which a type can be derived in a context. We extend the notion of a context to a partial function on \( V_{\text{cons}} \cup V_{\text{term}} \), which assigns kinds to constructor variables and types to term variables.

We write \( \Gamma \vdash M : \sigma \) if we can derive that in context \( \Gamma \) the term \( M \) has type \( \sigma \), using the following rules:

\[ \Gamma \vdash e^\sigma : \sigma \quad (e^\sigma \in C_{\text{term}}) \]
\[ \Gamma, x : \sigma \vdash z : \sigma \quad (z \in V_{\text{term}}) \]
\[ \Gamma, z : \sigma \vdash \lambda x : \sigma.M : \tau \quad (\lambda x : \sigma.M) : \sigma \to \tau \quad (\rightarrow I) \]
\[ \Gamma \vdash M : \sigma \Rightarrow \tau \quad \Gamma \vdash N : \sigma \quad (\rightarrow E) \]
Term equality is the equality induced by the $\beta$ and $\eta$ rules (for both term and type abstraction and application).

2 property

(i) the type of a term in a given context is unique (up to $\beta\eta$-equality)

(ii) equal terms have equal types

2.2 Semantics: general model definition

We now give the general structure of an environment model for second order lambda calculus, as described in [BMM90]. The difference is that types are interpreted as cpos whereas in [BMM90] types are interpreted as sets. Because terms may depend on types (in terms $M\sigma$), but types and other constructors cannot depend on terms, we can first consider the semantics of constructor expressions separately.

the semantics of constructor expressions

As we mentioned earlier, constructors (with their kinds) form a simple typed typed lambda calculus. So as the (sub)model for the constructor expressions we can take a model for the simple typed lambda calculus.

3 definition (environment model for constructor expressions)

An environment model for the constructor expressions over $V_{cons}$ and $C_{cons}$ is a 3-tuple $< Kind_{cons}, \Phi_{cons}, \mathcal{I}_{cons} >$, where

- $Kind = < Kind_\kappa | \kappa \text{ is a kind } >$ is a family of sets, indexed by kind expressions.
- $\Phi_{cons} = < \Phi_{\kappa_1 \Rightarrow \kappa_2} | \kappa_1 \Rightarrow \kappa_2 \text{ is a kind } >$ is a family of bijections such that
  $\Phi_{\kappa_1 \Rightarrow \kappa_2} \in Kind_{\kappa_1 \Rightarrow \kappa_2} \rightarrow [Kind_{\kappa_1} \rightarrow Kind_{\kappa_2}]$,
  where the square brackets denote some subset of the function space.
- $\mathcal{I}_{cons} \in C_{cons} \rightarrow \bigcup_\kappa Kind_\kappa$ gives the meanings of the constructor constants. Of course
  $\mathcal{I}_{cons}(c^\kappa) \in Kind_\kappa$ for all $c^\kappa \in C_{cons}$.

The meaning of a constructor expression of kind $\kappa$ will be a element of the set $Kind_\kappa$.

The bijections $\Phi_{\kappa_1 \Rightarrow \kappa_2}$ are the element-to-function mappings, well-known from models of the type-free lambda calculus. In fact, for the simple typed lambda calculus we do not need the $\Phi_{\kappa_1 \Rightarrow \kappa_2}$; we can take $Kind_{\kappa_1 \Rightarrow \kappa_2} = Kind_{\kappa_1} \rightarrow Kind_{\kappa_2}$ and all the $\Phi_{\kappa_1 \Rightarrow \kappa_2}$ the identity on $Kind_{\kappa_1} \rightarrow Kind_{\kappa_2}$.
We maintain the $\Phi_{\kappa_1\Rightarrow\kappa_2}$ here to emphasize the similarity with the definition of the semantics of terms that will be given later.

If we can derive $\Gamma \vdash \sigma : \kappa$, $[\Gamma \vdash \sigma : \kappa] \eta$ is the meaning of the constructor expressions $\sigma$ in environment $\eta$. Here an environment $\eta$ is a function which gives the meanings of the free constructor variables occurring in $\Gamma$, so $\eta \in \mathcal{V}_{cons} \rightarrow \bigcup \mathcal{K} ind_\kappa$.

We say that environment $\eta$ satisfies context $\Gamma$, written $\Gamma \models \eta$, if $\eta(\alpha) \in \mathcal{K} ind_\kappa$ for all $\alpha : \kappa$ in $\Gamma$.

For these environments we define the semantics of constructor expressions, by induction on their kind derivation, as follows:

\[
\begin{align*}
[\Gamma \vdash \alpha : \kappa] \eta &= \eta(\alpha) \\
[\Gamma \vdash c : \kappa] \eta &= \mathcal{I}_{cons}(c) \\
[\Gamma \vdash \sigma \tau : \kappa_2] \eta &= (\Phi_{\kappa_1\Rightarrow\kappa_2} [\Gamma \vdash \sigma : \kappa_1] \eta) \cdot [\Gamma \vdash \tau : \kappa_2] \eta \\
[\Gamma \vdash (\lambda \alpha : \kappa_1. \sigma) : \kappa_1 \Rightarrow \kappa_2] \eta &= \Phi_{\kappa_1\Rightarrow\kappa_2}^{-1}(\lambda \alpha \in \mathcal{K} ind_{\kappa_1}. [\Gamma, \alpha : \kappa_1 \vdash \sigma : \kappa_2] \eta[\alpha := a])
\end{align*}
\]

(1) (2) (3) (4)

Remember that every constructor has a unique kind, so there is only one possible choice for the kind $\kappa_1$ of $\sigma$ in (3). This guarantees that (3) defines a unique meaning for $\sigma \tau$.

For the semantics of construction expression to be defined correctly $\Phi_{\kappa_1\Rightarrow\kappa_2}$ has to be defined for all possible $\Gamma$ and $\sigma$. In other words, the range of the $\Phi_{\kappa_1\Rightarrow\kappa_2}$ must be large enough. In the actual models we will construct this will never be a problem. We will always have $\mathcal{K} ind_{\kappa_1\Rightarrow\kappa_2} = \mathcal{K} ind_{\kappa_1} \rightarrow \mathcal{K} ind_{\kappa_2}$, and $\Phi_{\kappa_1\Rightarrow\kappa_2}$ the identity on $\mathcal{K} ind_{\kappa_1\Rightarrow\kappa_2}$.

For this definition of a constructor model kind we can prove soundness,

\[
[\Gamma \vdash \rho : \kappa] \eta \in \mathcal{K} ind_\kappa,
\]

as well as soundness with respect to constructor equality,

\[
\Gamma \vdash \sigma =_e \tau : \kappa \Rightarrow [\Gamma \vdash \sigma : \kappa] \eta = [\Gamma \vdash \tau : \kappa] \eta
\]

(see [BMM90]).

the semantics of terms

The definition of the semantics of terms will be similar to the definition of the semantics of constructors.

Instead of having a family of sets $\mathcal{K} ind$, indexed by kinds, we will now need a family of cpos $\mathcal{D}om$, indexed by $\mathcal{K} ind_\kappa$. As for the constructor expressions, we can only talk about the meaning of terms in a context and a matching environment. The meaning of $\Gamma \vdash M : \sigma$ in an environment $\eta$ will be an element of $\mathcal{D}om_{[\Gamma \vdash \sigma \rightarrow \tau]. \eta}$.

To define the semantics of terms we will need mappings similar to the element-to-function mappings $\Phi_{\kappa_1\Rightarrow\kappa_2}$ we needed to define the semantics of constructors. However, because we have two kinds of abstraction, over term and over type variables, it will be slightly more complicated.

First we consider the function types.

Suppose $\Gamma \vdash M : \sigma \rightarrow \tau$. Then for all $\Gamma \vdash N : \sigma$ we have $\Gamma \vdash MN : \tau$, so we should be able to define the meaning of $MN (\in \mathcal{D}om_{[\Gamma \vdash \sigma \rightarrow \tau]. \eta})$ in terms of the meanings of $M (\in \mathcal{D}om_{[\Gamma \vdash \sigma \rightarrow \tau]. \eta})$
and $N \in \text{Dom}_N$. To get the meaning of $MN$, the meaning of $M$ has to be considered as a mapping from $\text{Dom}_M$ to $\text{Dom}_N$. So we require

$$\text{Dom}_M \cong [\text{Dom}_M \rightarrow \text{Dom}_N]$$

where the square brackets denote some subset of the function space.

The isomorphism corresponding with (i), the bijection

$$\phi_{\text{Dom}_M \rightarrow \text{Dom}_N} \in \text{Dom}_M \rightarrow [\text{Dom}_M \rightarrow \text{Dom}_N]$$

is the element-to-function mapping that we need to define the meaning of term abstraction and application.

For polymorphic types we need different mappings. Suppose $\Gamma \vdash M : \Pi \cdot f \cdot \tau$. Then for all $\tau$, $\Gamma \vdash M \cdot \tau : \Pi \cdot f \cdot \tau$. So we should be able to define the meaning of $M \cdot \tau \in \text{Dom}_{\text{Dom}}$ in terms of the meanings of $M$ and $\tau$, which are elements of $\text{Dom}_{\text{Dom}}$ and $\text{Kind}$, respectively. This is achieved by requiring

$$\text{Dom}_{\text{Dom}} \cong \prod_{\alpha \in \text{Kind}} \text{Dom}_{\text{Dom}}$$

where $\alpha$ is of course a fresh type variable.

The isomorphism corresponding with (ii), the bijection

$$\phi_{\text{Dom}_{\text{Dom}}} \in \text{Dom}_{\text{Dom}} \rightarrow \prod_{\alpha \in \text{Kind}} \text{Dom}_{\text{Dom}}$$

will be used to define the meaning of type abstraction and application.

We now have recursive domain equations for all function types and all polymorphic types. For the sake of a more uniform treatment, we also want a recursive domain equation for the remaining types, the base types. For every base type $\sigma$ a cpo $\text{domain}_\sigma$ has to be given. We could of course take $\text{Dom}_\sigma$ equal to $\text{domain}_\sigma$, but instead we will require

$$\text{dom}_\sigma \cong \text{domain}_\sigma$$

For all $\alpha \in \text{Kind}$, we define a function $F_\alpha$ that maps a family of cpos to a single cpo. If $\langle D_\alpha | \alpha \in \text{Kind} >$ is a family of cpos, then

$$F_\alpha \cong \text{domain}_\alpha$$

for base types $\sigma$

$$F_\alpha \cong [D_\alpha \rightarrow D_\alpha]$$

The system of coupled domain equations formed by (i), (ii) and (iii) can now be written as follows:

$$\forall \alpha \in \text{Kind} : \text{Dom}_\alpha \cong F_\alpha(\text{Dom})$$

We will now give the general model definition for second order environment models.
A second order environment model for \( A \) over \( V_{\text{term}} \), \( C_{\text{term}} \), \( V_{\text{cons}} \) and \( C_{\text{cons}} \) is a 6-tuple \( < \text{Kind}, \Phi_{\text{cons}}, I_{\text{cons}}, \text{Dom}, \Phi_{\text{term}}, I_{\text{term}} > \), where

- \( < \text{Kind}, \Phi_{\text{cons}}, I_{\text{cons}} > \) is an environment model for the constructor expressions over \( V_{\text{cons}} \) and \( C_{\text{cons}} \).

- \( \text{Dom} = < \text{Dom}_a \mid a \in \text{Kind}_a > \) is a family of cpos.

- \( \Phi_{\text{term}} = < \Phi_a \mid a \in \text{Kind}_a > \) is a family of continuous bijections such that
  \[ \Phi_a \in \text{Dom}_a \rightarrow F_a(\text{Dom}) \]
  where the \( F_a \) are defined by
  \[
  F_\text{r} \varphi \sigma \rightarrow \pi \eta < D_a \mid a \in \text{Kind}_a > = \text{domain}_\sigma \quad \text{for base types } \sigma \\
  F_\text{r} \alpha \rightarrow \tau \pi \eta < D_a \mid a \in \text{Kind}_a > = [D_\text{r} \varphi \sigma \rightarrow \tau \pi \eta] \\
  F_\text{r} \pi \varphi \sigma \rightarrow \tau \eta < D_a \mid a \in \text{Kind}_a > = \prod_{a \in \text{Kind}_a} D_\text{r} \alpha \rightarrow \tau \eta[a := a]
  \]

- \( I_{\text{term}} \in C_{\text{term}} \rightarrow \bigcup_{a \in \text{Kind}_a} \text{Dom}_a \) gives the meanings of the term constants. Of course \( I_{\text{term}}(c^\eta) \in \text{Dom}_\text{r} \varphi \sigma \rightarrow \pi \eta \) for all \( c^\eta \in C_{\text{term}} \).

- \( \eta \in (V_{\text{cons}} \cup V_{\text{term}}) \rightarrow (\bigcup_a \text{Kind}_a \cup \bigcup_a \text{Dom}_a) \). We say that an environment \( \eta \) satisfies a context \( \Gamma \), again written \( \Gamma \models \eta \), if \( \eta(a) \in \text{Kind}_a \) for all \( a : \sigma \) in \( \Gamma \) and \( \eta(x) \in \text{Dom}_\text{r} \varphi \sigma \rightarrow \pi \eta \) for all \( x : \sigma \) in \( \Gamma \).

For these environments we define the semantics of term expressions, by induction on their type derivation, as follows:

\[
\begin{align*}
\Gamma \vdash x : \sigma \eta &= \eta(x) \quad (1) \\
\Gamma \vdash c : \sigma \eta &= I_{\text{Term}}(c) \quad (2) \\
\Gamma \vdash M N : \tau \eta &= (\Phi_\text{r}, \Gamma \vdash M : \sigma \rightarrow \tau \eta) \quad (3) \\
\Gamma \vdash (\lambda x : \sigma.M) : \sigma \rightarrow \tau \eta &= \Phi_\text{r}^{-1}(\lambda \xi \in \text{Dom}_\text{r} \varphi \sigma \rightarrow \tau \eta \{ x := \xi \}) \quad (4) \\
\Gamma \vdash M \sigma : f \sigma \eta &= (\Phi_\text{r}, \Gamma, \alpha : \ast \vdash M : \Pi(\Lambda \alpha : \ast. \tau) \eta) \quad (5) \\
\Gamma \vdash (\Lambda \alpha : \ast.M) : \Pi(\Lambda \alpha : \ast. \tau) \eta &= \Phi_\text{r}^{-1}(\lambda a \in \text{Kind}_a. \Gamma, \alpha : \ast \vdash M : \tau \eta[a := a]) \quad (6) \\
\Gamma \vdash M : \sigma \eta &= \Gamma \vdash M : \tau \eta \quad (7)
\end{align*}
\]

Here \( s \) is \( \Gamma \vdash \sigma \rightarrow \tau : \ast \eta \) and \( t \) is \( \Gamma \vdash \Pi(\Lambda \alpha : \ast. \tau) : \ast \eta \).

We require that the ranges of the \( \Phi_a \) are large enough, so that the right-hand sides of (4) and (6), which involve a \( \Phi_\text{r}^{-1} \), are always defined.
There may be several derivations for $\Gamma \vdash M : \sigma$, but because terms have a unique type it can easily be proved that all type derivations give the same meaning. For this general model definition we can prove type soundness,

$$[\Gamma \vdash M : \sigma] \eta \in \text{Dom}(\Gamma \vdash \sigma \cdot \cdot)$$

as well as soundness with respect to term equality,

$$\Gamma \vdash M = N \Rightarrow [\Gamma \vdash M : \sigma] \eta = [\Gamma \vdash N : \sigma] \eta$$

(see [BMM90]).

2.3 The construction of a cpo model

In this section we will construct a cpo model for $A$.

First we consider the submodel for the constructor expressions. For this we can use a simple term model. So $[\Gamma \vdash \sigma : \alpha] \eta$ is the equivalence class of constructor expressions $\beta\eta$-equal to the expression obtained by substituting $\eta(\alpha)$ for $\alpha$ in $\eta$, for all free constructor variables $\alpha$.

**notation** To keep things readable, we write $\sigma \rightarrow \tau$ for $[\Gamma \vdash \sigma \rightarrow \tau] \eta$, $\Pi f$ for $[\Gamma \vdash \Pi f : \tau] \eta$ and $fa$ for $[\Gamma, \alpha : \tau \vdash f\alpha : \tau] \eta[\alpha := a]$. These abbreviations will be used throughout this paper, whenever we are dealing with a term model as the submodel for the constructor expressions. When we have a different constructor model, or when we are discussing a general model definition, we will write $[\sigma \rightarrow \tau]$, $[\Pi f]$ and $[f\alpha]$.

Because of the general model definition we have given, there only remains the task of finding a family of cpos $\text{Dom} = \langle \text{Dom}_{\alpha} | \alpha \in \text{Kind} \rangle$, that solves the system of coupled domain equations:

$$\forall \alpha \in \text{Kind} : \quad \text{Dom}_{\alpha} \cong F_{\alpha}(\text{Dom}) \quad (i)$$

with the associated continuous bijections $\Phi_{\alpha} \in \text{Dom}_{\alpha} \rightarrow F_{\alpha}(\text{Dom})$.

We use a standard technique, described in [SP82], to find a solution for the recursive domain equations. For this some category theory is needed. A clear and self-contained presentation of this technique can be found in [BH88].

An $\omega$-category is a category with an initial object in which every $\omega$-chain has a colimit. A functor is called $\omega$-continuous if it preserves colimits of $\omega$-chains. A fixed point of a functor $F : \mathcal{K} \rightarrow \mathcal{K}$ is a pair $(D, \phi)$, where $D$ is a $\mathcal{K}$-object and $\phi$ an isomorphism between $D$ and $F(D)$.

The initial fixed point theorem ([SP82],[BH88]) states that for an $\omega$-continuous functor on an $\omega$-category an initial fixed point can be constructed, rather like for every continuous function on a cpo a least fixed point can be constructed. In fact, the fixed point theorem for cpos is a particular case of the initial fixed point theorem for $\omega$-categories.

This result can be used to find a solution of a recursive domain equation. Because of the interdependence of the domain equations, we have to solve all of them together. We consider one recursive domain equation $D \cong F(D)$, where $D$ is a family of cpos. We will construct a solution in a product category.
product categories

Let $I$ be an index set and $C$ a category. The product category $K = \prod_{a \in I} C$ is then defined as follows:

- objects of $K$ are families $< D_a | a \in I >$, where each $D_a$ is a $C$-object
- a $K$-morphism from $< D_a | a \in I >$ to $< E_a | a \in I >$ is a family $< f_a | a \in I >$, where each $f_a$ is a $C$-morphism from $D_a$ to $E_a$.

If $C$ is an $\omega$-category, then so is $\prod_{a \in I} C$ (see [HS73], [tEH89b]).

For every $b \in I$ we have a projection functor $P_b$ from $K$ to $C$, which selects the $b$-component of a $K$-object or morphism, i.e. $P_b(< X_a | a \in K^I_a >) = X_b$. The projection functors are $\omega$-continuous (see [tEH89b]).

A functor $F$ from $K$ to $K$ can be considered as a family of functors $< F_a | a \in I >$, where every $F_a$ is a functor from $K$ to $C$. It is easily shown that $F$ is $\omega$-continuous if every component $F_a$ is $\omega$-continuous (see [tEH89b]).

Tupling of functors will be denoted by $< , >$. For example, $< P_a, P_b > : K \to C \times C$ is the functor which selects the $a$ and $b$ components of a $K$-object or morphism.

the construction of a second order model

$CPO$ is the category with cpos as objects and continuous functions as morphisms.

For the domain equations for function types we have the function space functor, $FS$, defined by

- $FS : CPO^{OP} \times CPO \to CPO$
- if $D$ and $E$ are cpos, then $FS(D, E) = [D \to E]$, the cpo of continuous functions from $D$ to $E$, with the ordering pointwise.
- if $f \in [D' \to D]$ and $g \in [E \to E']$, then $FS(f, g) = (\lambda x \in [D \to E]. g \circ f \circ x) \in [[D \to E] \to [D' \to E']]$

For the polymorphic types we have the generalized product functor, $GP$, defined by

- $GP : \prod_{a \in I} CPO \to CPO$
- if $< D_a | a \in I >$ is a family of cpos, then $GP(< D_a | a \in K^I_a >) = \prod_{a \in I} D_a$, the cpo which is the product of all the cpos $D_a$, with the ordering coordinatewise.
- if $< f_a | a \in I >$ is a family of functions, where $f_a \in [D_a \to E_a]$ for all $a \in I$, then $GP(< f_a | a \in I >) = \lambda (g \in [D \to E]. f_a \circ g) \in GP(< E_a | a \in I >)$. $< f_a(d_a) | a \in I >$ which is a continuous function from $GP(< D_a | a \in I >)$ to $GP(< E_a | a \in I >)$.

Because of the contravariance of $FS$ in its first argument we cannot solve the recursive domain equations in the category $\prod_{a \in K^I_a} CPO$.

This problem is overcome using the standard technique. In [SP82] a theory of $O$-categories, a special class of categories, is developed. For an $O$-category $C$ there is an associated category of embedding-projection pairs $C_{PR}$, and given a functor $F$ on an $O$-category $C$, a corresponding functor $F_{PR}$ on the category $C_{PR}$ can be defined, which is covariant in all its arguments.
CPO is an O-category. The associated category of embedding-projection pairs is CPOR.

CPOR is the category with cpos as objects and embedding-projection pairs as morphisms. An embedding-projection pair from cpo A to cpo B is a pair (ϕ, ψ) of continuous functions ϕ : A → B and ψ : B → A such that ψϕ = idA and ϕψ ⊆ idB.

CPOR is an w-category (see [SP82], [BH88]).

The functors corresponding with FS and GP are FSOR : CPOR × CPOR → CPOR and GPOR : ∏α∈Kind. CPOR → CPOR. They are defined as follows

FSOR(D, E) = FS(D, E)
FSOR((ϕ, ψ), (ϕ', ψ')) = (FS(ϕ, ψ'), FS(ϕ', ψ'))

and

GPOR(<Da | a ∈ I>) = GP(<Da | a ∈ I>)
GPOR(<ϕa, ψa | a ∈ I>) = (GP(<ϕa | a ∈ I>), GP(<ψa | a ∈ I>))

So the object parts are unchanged. FSOR and GPOR are w-continuous (see [SP82] or [BH88] for FSOR, and [IEH88b] for GPOR).

For the base types we will need constant functors. If A is a cpo then CA : K → CPOR is the functor which maps every K-object to the cpo A, and every K-morphism to the identity morphism on A, which in the category CPOR is the embedding-projection pair ((A, A, O, A, ~), (A, A, ~)).

We will construct Dom in the product category K = ∏α∈Kind. CPOR.

We define F : K → K by

F = <Fa | a ∈ Kind. >

where the functors Fa : K → CPOR are defined as follows

Fa = Cdomaina
Fa × T = FSOR × <Pα, Pα >
FαT = GPOR × <Pαa | a ∈ Kind. >

Since FSOR, GPOR, CA and Pa are all ω-continuous, so are all the Fa and hence so is F. Then by the initial fixed point theorem an initial fixed point can be constructed.

Let (Dom, m) be a fixed point of F. Then m is an isomorphism from Dom to F(Dom) in ∏ CPOR. Because everything is defined pointwise, this means that all its components ma = (Φa, Ψa) are isomorphisms from Domα to Fα(Dom) in CPOR (i.e. Ψa = Φa⁻¹). So Dom solves the recursive domain equations, and the embeddings Φa : Domα → Fa(Dom) are the bijections we need.

So an initial fixed point of F gives a family of cpos Dom that satisfies the recursive domain equations and the associated bijections.

So, recapitulating,

- CPOR is an ω-category
- ∏α∈Kind. CPOR is an ω-category
- FSOR, GPOR, CA and Pa are ω-continuous
- for all α ∈ Kind. the functor Fa : ∏ CPOR → CPOR is ω-continuous
• the functor $F = \langle F_a | a \in Kind \rangle : \Pi_{\text{CPO}_{PR}} \rightarrow \Pi_{\text{CPO}_{PR}}$ is $\omega$-continuous

• in $\Pi_{a \in Kind.} \text{CPO}_{PR}$ the equation $D \cong F(D)$ has an initial solution $(\text{Dom}, m)$ where $\text{Dom} = \langle \text{Dom}_a | a \in Kind \rangle$ and $m = \langle m_a | a \in Kind \rangle$

• $m_a = (\Phi_a, \Psi_a)$ is an isomorphism between $\text{Dom}_a$ and $F_a(\text{Dom})$ for all $a \in Kind$.  

3 Recursive types

In this section we consider the extension of second order lambda calculus with recursive types. As far as the constructors expressions are concerned, we just add another constructor constant \( \mu \), of kind \( (* \Rightarrow *) \Rightarrow * \). So if \( \Gamma \vdash f : * \Rightarrow * \), then \( \Gamma \vdash \mu f : * \).

The whole idea behind a recursive type \( \mu f \) is that it is a solution of

\[
\mu f \approx f(\mu f)
\]

so that the types \( \mu f, f(\mu f), f(f(\mu f)), \ldots \) are equivalent.

For example, if we have an expression \( M \) of type \( \mu f \), where \( f \equiv (\Lambda \alpha : \ast. \alpha \rightarrow \text{int}) \). Because of the equivalence between \( \mu f \) and \( f(\mu f) = \mu f \rightarrow \text{int} \), we want to be able to apply \( M \) to itself, and the result should be of type \( \text{int} \).

This means that for all \( \Gamma \vdash f : * \Rightarrow * \) we require that

\[
\text{Dom}_{\Gamma \vdash \mu f : *} \equiv \text{Dom}_{\Gamma \vdash f(\mu f) : *} \quad (i)
\]

We will consider three ways to treat recursive types:

A\( \mu_1 \) A recursive type \( \mu f \) and its unfolding \( f(\mu f) \) are not identified.

Because we want terms to have a unique type, this means that we cannot both have \( \Gamma \vdash M : \mu f \) and \( \Gamma \vdash M : f(\mu f) \). We introduce explicit coercion operators \( \text{fold}_{\mu f} \) and \( \text{unfold}_{\mu f} \) in the syntax of terms. If \( \Gamma \vdash M : \mu f \) then \( \Gamma \vdash \text{unfold}_{\mu f} M : f(\mu f) \), and if \( \Gamma \vdash M : f(\mu f) \) then \( \Gamma \vdash \text{fold}_{\mu f} M : \mu f \). The meaning of the fold and unfold operators is given by the isomorphism between \( \text{Dom}_{\Gamma \vdash \mu f : *} \) and \( \text{Dom}_{\Gamma \vdash f(\mu f) : *} \).

A\( \mu_2 \) A recursive type \( \mu f \) and its unfolding \( f(\mu f) \) are identified.

So \([\Gamma \vdash \mu f : *] \eta = [\Gamma \vdash f(\mu f) : *] \eta = [\Gamma \vdash f(f(\mu f)) : *] \eta = \ldots\), which means (i) is trivially satisfied, and if \( \Gamma \vdash M : \mu f \) then also \( \Gamma \vdash M : f(\mu f) \) and vice versa.

A\( \mu_3 \) We interpret recursive types as infinite types. This means we not only identify recursive types and their unfoldings, but that we identify all recursive types that have the same infinite unfolding.

For example, the types \( \mu (\Lambda \alpha : \ast. \alpha \rightarrow \text{int}) \) and \( \mu (\Lambda \alpha : \ast. (\alpha \rightarrow \text{int}) \rightarrow \text{int}) \), will be identified, because if we keep unfolding them they both have the same "limit", namely \((((\ldots) \rightarrow \text{int}) \rightarrow \text{int}) \rightarrow \text{int}) \rightarrow \text{int} \).

In A\( \mu_2 \) these types would not be identified, because by unfolding them we can never get the same term: unfoldings of the first type will be of the form \( ((\ldots(\mu (\Lambda \alpha : \ast. \alpha \rightarrow \text{int}) \ldots) \rightarrow \text{int}) \rightarrow \text{int}) \rightarrow \text{int} \)

and unfoldings of the second type will be of the form \( ((\ldots(\mu (\Lambda \alpha : \ast. (\alpha \rightarrow \text{int}) \rightarrow \text{int}) \ldots) \rightarrow \text{int}) \rightarrow \text{int}) \rightarrow \text{int} \)

In the next three sections we will consider how for each of these systems the general model definition and the construction of the cpo model given in part 2 are affected. The general model definition will be changed for each system, and we will alter the construction of the cpo model accordingly.
3.1 $\Lambda \mu_1$

3.1.1 Syntax

constructors

The definition of constructor expressions is unchanged. We just have a new constructor constant, $\mu : (\ast \Rightarrow \ast) \Rightarrow \ast$, for making recursive types.

Constructor equality is $\beta\eta$-equality.

terms

The set of pseudo-terms over $C_{term}$ and $V_{term}$ is now defined by

$$M = c \mid x \mid (\lambda x : \sigma. M) \mid M_1 M_2 \mid (\lambda \alpha : \ast. \sigma) \mid M \sigma \mid \text{fold}_{M} M \mid \text{unfold}_{M} M$$

where $c \in C_{term}$, $x \in V_{term}$, and $\sigma$ and $f$ are pseudo-constructors.

We have two additional type inference rules

$$\frac{}{\Gamma \vdash M : \mu f} (UNFOLD) \quad \text{and} \quad \frac{}{\Gamma \vdash \text{fold}_{M} M : \mu f} (FOLD)$$

As remarked in [tEM88], the subscript $\mu f$ of $\text{fold}_{M}$ is necessary. If it is omitted, some terms no longer have a unique type. This is shown in the following example.

Suppose $\Gamma \vdash M : f(\mu f)$. Using $(FOLD)$ we can derive two types for $\text{fold}_{M}$: the type $\mu f$ of course, but also the type $\mu g$, $g = (\lambda \alpha : \ast. f(\mu f))$, where $\alpha$ is a fresh type variable. Since $\alpha$ does not occur in $f(\mu f)$, $f(\mu f) = f(\mu f) [\alpha := \mu g] = g(\mu g)$.

We needed the fact that terms have a unique type to guarantee that the definition of the meaning of a term was unambiguous. Therefore $\text{fold}$ is written with the subscript $\mu f$. By symmetry, we also write $\text{unfold}$ with a subscript $\mu f$. This subscript, however, could be omitted without causing any problems.

We redefine term equality for $\Lambda \mu_1$. Term equality is the congruence relation generated by $\beta\eta$-equality and the following two rules:

$$\text{fold}_{M} (\text{unfold}_{M} M) = M$$
$$\text{unfold}_{M} (\text{fold}_{M} M) = M$$

3.1.2 Semantics: general model definition

The definition of the semantics of constructor expressions can remain unchanged, since we have no new rules for kind derivations.

We do have new rules for the type inference system. We have to define the meaning of terms that are typed using the new type inference rules $(FOLD)$ and $(UNFOLD)$.

For this we require

$$\text{Dom}_{\Gamma \vdash \mu f, \ast} \cong \text{Dom}_{\Gamma \vdash f(\mu f), \ast} \cong$$

16
The associated isomorphism, the bijection
\[ \Phi_{\Gamma^\mu(f)} : \text{Dom}(\Gamma^\mu(f)) \rightarrow \text{Dom}(\Gamma^\mu(f)) \]
gives the semantics of folding and unfolding:
\[ \begin{align*}
[\Gamma \vdash \text{unfold}_\mu M : f(\mu f)] \eta &= \Phi_{\Gamma^\mu(f)}([\Gamma \vdash M : \mu f] \eta) \\
[\Gamma \vdash \text{fold}_\mu M : \mu f] \eta &= \Phi_{\Gamma^\mu(f)}^{-1}([\Gamma \vdash M : f(\mu f)] \eta)
\end{align*} \]
We extend the definition of \( F_a | a \in \text{Kind}_* \) with
\[ F_{\Gamma^\mu(f)}(\text{Dom} \, | \, a \in \text{Kind}_*) = \text{Dom}(\Gamma^\mu(f)) \]

5 Definition (general model definition \( A\mu_1 \))
An environment model for \( A\mu_1 \) is defined as for \( A \) (definition 4), except with the definition of \( \Gamma \) extended as above, and with \( F = F_a | a \in \text{Kind}_* \) defined by
\[
\begin{align*}
F_{\sigma}(\text{Dom} \, | \, a \in \text{Kind}_*) &= \text{dom}_{\sigma} \quad \text{for base types } \sigma \\
F_{\sigma \rightarrow \tau}(\text{Dom} \, | \, a \in \text{Kind}_*) &= D_{(\sigma \rightarrow \tau)} \\
F_{\Pi f}(\text{Dom} \, | \, a \in \text{Kind}_*) &= \Pi_{f \sigma \in \text{Kind}_*} \text{Dom} \, | \, \sigma \in \text{Kind}_* \\
F_{\mu f}(\text{Dom} \, | \, a \in \text{Kind}_*) &= \text{Dom}(\Gamma^\mu(f))
\end{align*}
\]

3.1.3 The construction of a cpo model
The definition for the submodel for the constructors is the same as it was for \( A \), so we can use the same submodel we used for \( A \), i.e. a term model.
To complete the model we have to construct a family of cpos \( \text{Dom} \, | \, a \in \text{Kind}_* \), that solves the system of coupled domain equations
\[ \forall a \in \text{Kind}_* : \text{Dom} \, | \, a \in \text{Kind}_* \cong F_a(\text{Dom}) \]
We have the additional domain equations
\[ \text{Dom}(\Gamma^\mu(f)) \cong \text{Dom}(\Gamma^\mu(f)) \]
for all \( \Gamma \vdash f : * \Rightarrow * \).
Now we have seen how we found a solution of the system of coupled domain equations for \( A \), solving the new system of coupled domain equations for \( A\mu_1 \) is completely straightforward.
We define a new functor \( F : \mathcal{K} \rightarrow \mathcal{K} \) by \( F = F_a | a \in \text{Kind}_* \), where the \( F_a : \mathcal{K} \rightarrow \text{CPO}_{P_R} \) are defined as follows
\[
\begin{align*}
F_{\sigma} &= \text{C}_{\text{dom}_{\sigma}} \quad \text{for base types } \sigma \\
F_{\sigma \rightarrow \tau} &= F_{\text{SP} \sigma} < F_{\tau}, F_{\tau} > \\
F_{\Pi f} &= \text{GP} \sigma \sigma < F_{\sigma}, F_{\alpha} | a \in \text{Kind}_* > \\
F_{\mu f} &= F_{\text{P}_f(f)}
\end{align*}
\]
The initial fixed point of \( F \) gives the cpos \( \text{Dom}_a \) satisfying the recursive domain equations, and the associated isomorphisms \( \Phi_a \in \text{Dom}_a \rightarrow F_a(\text{Dom}) \).
3.2 $A\mu_2$

3.2.1 Syntax

We define constructor equality, $=_c$, as the equivalence relation induced by $\beta\eta$-equality and by

$$\Gamma \vdash \mu f =_c f(\mu f) \quad \text{for all } \Gamma \vdash f : * \Rightarrow *$$

We call the equality induced by the rule above $\mu$-equality.

Using the type conversion rule

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash \sigma =_c \tau}{\Gamma \vdash M : \tau} \quad (TEQ)$$

we can derive

$$\frac{\Gamma \vdash M : \mu f \quad \Gamma \vdash M : f(\mu f)}{\Gamma \vdash M : f(\mu f)}$$

So we can drop the fold$_{\mu f}$ and unfold$_{\mu f}$ from our syntax, and we no longer need the extra domain equations

$Dom[\Gamma;\mu f;\downarrow] \cong Dom[\Gamma;f(\mu f);\downarrow]$ we needed for $A\mu_1$, since $\mu f =_c f(\mu f)$, and so $[\Gamma \vdash \mu f : *] \eta = [\Gamma \vdash f(\mu f) : *] \eta$.

3.2.2 Semantics: general model definition

We can take the same recursive domain equations we had for $A$:

$$\forall a \in Kind. : \quad Dom_a \cong F_a(Dom)$$

where

$$F_{\sigma}(< D_a | a \in Kind. >) = \text{domain}_\sigma$$
$$F_{\sigma \rightarrow \tau}(< D_a | a \in Kind. >) = [D_{\sigma \rightarrow \tau} \rightarrow D_{\tau}]$$
$$F_{\mu f}(< D_a | a \in Kind. >) = \prod_{a \in \text{Kind.}} D_{\mu f(a)}$$

For the recursive types these domain equations achieve precisely what we want them to. Because $\mu f =_c f(\mu f)$, the constructor model should satisfy $[\Gamma \vdash \mu f : *] \eta = [\Gamma \vdash f(\mu f) : *] \eta$ and for the recursive type $\mu f \equiv \mu(\Lambda \alpha : \ast \cdot \alpha \rightarrow \text{int})$ we then get

$$Dom[\mu f] = Dom[\mu f \rightarrow \text{int}$$

, since $[\mu f] \equiv [f(\mu f)] \equiv [\mu f \rightarrow \text{int}]$

$\cong [Dom[\mu f] \rightarrow Dom[\text{int}]]$

$\cong [Dom[\mu f \rightarrow \text{int}] \rightarrow Dom[\text{int}]]$

$\cong \ldots$

If $\mu f =_{\beta\eta\mu} \mu(\Lambda \alpha : \ast \cdot \alpha)$ then $F_{\mu f}$ is as yet undefined. We take

$$F_{\mu(\Lambda \alpha : \ast \cdot \alpha)}(< D_a | a \in Kind. >) = D_{\mu(\Lambda \alpha : \ast \cdot \alpha)}$$

This means the domain equation for $\mu(\Lambda \alpha : \ast \cdot \alpha)$ is

$$Dom[\mu(\Lambda \alpha : \ast \cdot \alpha)] \cong F_{\mu(\Lambda \alpha : \ast \cdot \alpha)}(< Dom[\Gamma;\alpha;\downarrow] | a \in Kind. >) = Dom[\mu(\Lambda \alpha : \ast \cdot \alpha)]$$
6 definition (general model definition \( \Lambda \mu_2 \))

An environment model for \( \Lambda \mu_2 \) is defined as for \( \Lambda \) (definition 4), except with \( F =< F_a | a \in \text{Kind}_a > \) defined by

\[
\begin{align*}
F_\sigma &= \text{domain}_\sigma, & \text{for base types } \sigma \\
F_{\sigma \rightarrow \tau} &= [D_\sigma] \rightarrow D_\tau \\
F_{\Pi \phi} &= \prod_{\phi \in \text{Kind}_a} D_\phi \\
F_{\mu(\lambda \alpha \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \......
3.3 $\Lambda\mu_3$

3.3.1 Syntax

We shall now interpret recursive types as infinite types.
To define the resulting congruence relation on types, we define a tree $T(\sigma)$ for every type $\sigma$. These trees will be \textit{regular} trees, i.e. trees with a finite set of subtrees. The leaves will be base types or types only consisting of constructor variables, and the nodes correspond to type constructors.

7 definition ($T$)
The function $T$ from types to regular trees is defined by

\[
T(\sigma) = \sigma \quad \text{if } \sigma \text{ consists only of constructor variables}
\]

\[
T(c) = c
\]

\[
T(\sigma \rightarrow \tau) = \begin{array}{c}
T(\sigma) \\
\downarrow
\end{array} T(\tau)
\]

\[
T(\Pi(\Lambda \alpha : \ast. \sigma)) = \Pi_{\alpha} \bigg\downarrow \bigg\uparrow T(\sigma)
\]

\[
T(\mu(\Lambda \alpha : \ast. \sigma)) = \begin{cases}
T(\sigma)[\alpha := T(\mu(\Lambda \alpha : \ast. \sigma))] & \text{if } T(\sigma) \neq \alpha \\
\bot & \text{if } T(\sigma) = \alpha
\end{cases}
\]

\[
T(\sigma) = T(\tau) \quad \text{if } \sigma =_{\beta_n} \tau
\]

Note that we have bound type variables in the trees: every $\Pi$-node introduces a bound type variable. $\alpha$-equal trees are identified. $T(\sigma)[\alpha := \ldots]$ is tree substitution.

By the following property it is clear that this defines a regular tree for every type.

8 property ([Cou83], theorem 4.2.1)
If $t \neq \alpha$, and $t$ is regular, then there is a unique tree $x$ such that $x = t[\alpha := x]$, and $x$ will be regular. □

Some examples. Suppose $\Gamma \equiv g : \ast \Rightarrow \ast, \, \beta : \ast$. Then

\[
T(\beta) = \beta
\]

\[
T(g\beta) = g\beta
\]

\[
T(\Pi(\Lambda \alpha : \ast. \alpha \rightarrow \beta)) = \Pi_{\alpha} \bigg\downarrow \bigg\uparrow \alpha \quad \beta
\]

\[
T(\mu(\Lambda \alpha : \ast. \alpha)) = \bot
\]

Let $f \equiv (\Lambda \alpha : \ast. \alpha \rightarrow \beta)$. Then

\[
T(\mu f) = \begin{array}{c}
\alpha \\
\downarrow
\end{array} \beta
\]

$[\alpha := T(\mu f)]$

\[
= \begin{array}{c}
T(\mu f) \\
\downarrow
\end{array} \beta
\]

20
So

\[ T(\mu f) = \]

\[ \beta \rightarrow \beta \]

We would get the same tree for \( \mu(\Lambda \alpha : \ast.(\alpha \rightarrow \beta) \rightarrow \beta), \mu(\Lambda \alpha : \ast.((\alpha \rightarrow \beta) \rightarrow \beta)) \), etc.

9 definition \((\approx)\)
The equivalence relation \(\approx\) on types is defined by

\[ \sigma \approx \tau \iff T(\sigma) = T(\tau) \]

\[ \square \]

For all types \(\sigma\) and \(\tau\), \(\sigma \approx \tau\) is decidable. (This is because \(T(\sigma)\) and \(T(\tau)\) are regular.) We take \(\approx\) as our notion of type equality. Equality for constructor expressions of higher kinds is the congruence relation generated by \(\approx\) and the \(\eta\) rules. So the type conversion rule \((TEQ)\) has become

\[ \Gamma \vdash M : \sigma \quad \sigma \approx \tau \]

\[ \Gamma \vdash M : \tau \]

3.3.2 Semantics: general model definition

Again we take the same recursive domain equations as for \(A\), and for the type \(\mu(\Lambda \alpha : \ast.\alpha)\) we add

\[ Dom_{\mu(\Lambda \alpha : \ast.\alpha)} = Dom_{\mu(\Lambda \alpha : \ast.\alpha)} \]

as we did for \(A\mu_3\).

10 definition \((\text{general model definition } A\mu_3)\)

An environment model for \(A\mu_3\) is defined as for \(A\) (definition 4), except with

\[ F = \Gamma \mid a \in Kind_a > \]

defined by

\[ F_\sigma[D_a | a \in Kind_a >] = \text{domain}_\sigma \quad \text{for base types } \sigma \]

\[ F_{\sigma \rightarrow \tau}[D_a | a \in Kind_a >] = [D_\sigma \rightarrow D_\tau] \]

\[ F_{\Pi \star}[D_a | a \in Kind_a >] = \prod_{a \in Kind_a} D_\star \]

\[ F_{\mu(\Lambda \alpha : \ast.\alpha)}[D_a | a \in Kind_a >] = D_{\mu(\Lambda \alpha : \ast.\alpha)} \]

\[ \square \]

3.3.3 The construction of a cpo model

the submodel for the constructor expressions

For \(A\mu_3\) we choose a different constructor model: types will be interpreted as trees. The leaves will be base types or type variables, and the nodes correspond to type constructors.

If all free constructor variables in \(\sigma\) are type variables, then the meaning of a type \(\sigma\) in environment \(\eta\) will be the tree

\[ T(\sigma)[\alpha_0 := \eta(a_0), \ldots, \alpha_n := \eta(a_n)] \]

i.e. \(T(\sigma)\) with all type variables \(\alpha\) replaced by \(\eta(\alpha)\).
For example

\[
\begin{align*}
[\Gamma \vdash \text{int} : * ] \eta &= \text{int} \\
[\Gamma, \alpha : * \vdash \alpha \rightarrow \text{int} : * ] \eta &= \eta(\alpha) \quad \rightarrow \text{int}
\end{align*}
\]

Let \( f \equiv (\lambda \alpha : * . \alpha \rightarrow \beta) \). Then

\[
[\Gamma \vdash \mu f : * ] \eta =
\]

\[
\quad \begin{array}{c}
\eta(\beta) \\
\eta(\beta)
\end{array}
\]

\( \text{Kind}_{\Rightarrow} \) will be a subset of \( \text{Kind} \rightarrow \text{Kind}_{\Rightarrow} \), and \( [\Gamma \vdash \mu f : * ] \eta \in \text{Kind}_{\Rightarrow} \) will be a fixed point of \( [\Gamma \vdash f : * \Rightarrow * ] \eta \in \text{Kind}_{\Rightarrow} \).

For instance, the meaning of \( f \) will be:

\[
[\Gamma \vdash f : * \Rightarrow * ] \eta = \lambda a \in \text{Kind}_{\Rightarrow}, [\Gamma \vdash \alpha : \beta \rightarrow * ] \eta[\alpha := a] \\
= \lambda a \in \text{Kind}_{\Rightarrow} \\
[\Gamma \vdash \alpha : * ] \eta[\alpha := a] \quad [\Gamma \vdash \beta : * ] \eta \\
= \lambda a \in \text{Kind}_{\Rightarrow} \\
\eta(\beta)
\]

So \( [\Gamma \vdash \mu f : * ] \eta = [\Gamma \vdash f : * \Rightarrow * ] \eta \quad [\Gamma \vdash \mu f : * ] \eta \).

We will now define the submodel for the constructors in the way prescribed by the general model definition. So we have to define \( \text{Kind} = \{ \text{Kind}_{\Rightarrow} | \kappa \text{ a kind} \} \), and we have to define \( \mathcal{I}_{\text{con}_{\Rightarrow}} \), giving the meaning of the constructor constants, with \( \mathcal{I}_{\text{con}_{\Rightarrow}}(c^\kappa) \in \text{Kind}_{\Rightarrow} \).

**11 definition (Tree)**

Tree is the set of all finite and infinite trees with base types and type variables as leaves, and \( \rightarrow \) and \( \Pi_\alpha \), \( \alpha \) a type variable, as nodes. \( \rightarrow \) nodes have two subtrees, \( \Pi_\alpha \) nodes have one subtree.

\( \square \)

We will define a partial order \( \sqsubseteq \) on \( \text{Tree} \), so that for all \( \Gamma \vdash f : * \Rightarrow * \) we can define \( [\Gamma \vdash \mu f : * ] \eta \) as the least fixed point of \( [\Gamma \vdash f : * \Rightarrow * ] \eta \).

**12 definition (\( \sqsubseteq \))**

\( \sqsubseteq \) is a partial order on \( \text{Tree} \) defined by

\[
\begin{align*}
\perp & \sqsubseteq a & \text{for all } a \in \text{Tree} \\
a & \sqsubseteq a & \text{for all } a \in \text{Tree} \\
& \quad \begin{array}{c}
\quad \sqsubseteq \\
\quad \quad \text{if } a \sqsubseteq a' \text{ and } b \sqsubseteq b'
\end{array} \\
\Pi_\alpha & \sqsubseteq \Pi_\alpha & \text{if } a \sqsubseteq b \\
\downarrow & \quad \downarrow \\
a & \downarrow b
\end{align*}
\]

\( \square \)
So a ⊆ b, if we can get a by cutting of some subtrees of b and replacing them by ⊥. Every ascending chain in Tree has a least upper bound.

<Kind₅ | κ a kind > is defined by

\begin{align*}
\text{Kind}_a &= \{ t | t \in \text{Tree} \land FV(t) = \emptyset \} \\
\text{Kind}_a &\rightarrow \ast = [\text{Kind}_a \rightarrow \text{Kind}_a] \\
\text{Kind}_{\ast \rightarrow \ast} &\rightarrow \ast = \text{Kind}_{\ast \rightarrow \ast}, \kappa_1 \Rightarrow \kappa_2 \neq \ast \Rightarrow \ast
\end{align*}

Here \( FV(t) \) denotes the set of free type variables occurring in \( t \).

\text{Kind}_a is restricted to functions from \( \text{Kind}_a \) to \( \text{Kind}_a \) that are continuous with respect to \( \subseteq \), because for all \( F \in \text{Kind}_{\ast \rightarrow \ast} \) we want \( \int [f : \ast \Rightarrow \ast \Rightarrow \mu f : \ast ] [f := F] \) to be the initial fixed point of \( F \). For \( \text{Kind}_{\ast \rightarrow \ast}, \kappa_1 \Rightarrow \kappa_2 \neq \ast \Rightarrow \ast \), we have no such requirements.

The meaning of the constructor constants is given by

\[ I_{\text{const}}(\sigma) = \sigma \quad \text{for base types} \quad \sigma \]
\[ I_{\text{const}}(\rightarrow) = p a \in \text{Kind}_a, a b \in \text{Kind}_a \]
\[ I_{\text{const}}(\Pi) = a F \in \text{Kind}_{\ast \rightarrow \ast}, \Pi \]
\[ I_{\text{const}}(\mu) = a F \in \text{Kind}_{\ast \rightarrow \ast}, \bigcup_{i \in \mathbb{N}} F^i \]

It is easy to see that \( I_{\text{const}}(\sigma) \in \text{Kind}_a, I_{\text{const}}(\rightarrow) \in \text{Kind}_{\ast \rightarrow (\ast \rightarrow \ast)} \), \( I_{\text{const}}(\Pi) \in \text{Kind}_{(\ast \rightarrow \ast) \rightarrow \ast} \) and \( I_{\text{const}}(\mu) \in \text{Kind}_{(\ast \rightarrow \ast) \rightarrow \ast} \).

The sets \( \text{Kind}_a \) are actually larger than they have to be. The following \( \text{Kind}_a \) could also be used

\[ \text{Kind}_a = \{ \text{t} | \text{t} \in \text{Tree} \land FV(t) = \emptyset \land \text{t is regular} \} \]
\[ \text{Kind}_{\ast \rightarrow \ast} = \{ (p a \in \text{Kind}_a) | \text{t} \in \text{Tree} \land FV(t) \subseteq \{a\} \land \text{t is regular} \} \]
\[ \text{Kind}_{\ast \rightarrow \ast}, \kappa_1 \rightarrow \ast = \text{Kind}_{\ast \rightarrow \ast}, \kappa_1 \Rightarrow \kappa_2 \neq \ast \Rightarrow \ast \]

Clearly \( I_{\text{const}}(\sigma) \in \text{Kind}_a, I_{\text{const}}(\rightarrow) \in \text{Kind}_{\ast \rightarrow (\ast \rightarrow \ast)} \) and \( I_{\text{const}}(\Pi) \in \text{Kind}_{(\ast \rightarrow \ast) \rightarrow \ast} \).

That \( I_{\text{const}}(\mu) \in \text{Kind}_{(\ast \rightarrow \ast) \rightarrow \ast} \) is an immediate consequence of the following lemma.

13 lemma For \( F \in \text{Kind}_{\ast \rightarrow \ast} \), \( \bigcup_{i \in \mathbb{N}} F^i \) \in \( \text{Kind}_a \)

proof
Suppose \( F = (p a \in \text{Kind}_a), t \in \text{Kind}_{\ast \rightarrow \ast} \). This means \( F' = t[a := t'], \) so
\[ F' = t[a := a], \ F'' = t[a := F'], \ F'^{i} = t[a := F'^{i}], \ldots \]
and therefore \( \bot \subseteq F'^{i} \subseteq F'^{i+1} \subseteq \ldots \). This chain has a lub, \( \bigcup_{i \in \mathbb{N}} F'^{i} \in \text{Tree}. \)

There remains to be shown that \( \bigcup F'^{i} \subseteq \text{Kind}_a \), i.e. that \( \bigcup F'^{i} \) contains no free variables, and that \( \bigcup F'^{i} \) is regular.

Clearly \( \bigcup F'^{i} \) does not contain free variables, since all \( F'^{i} \in \text{Kind}_a \), so none of them contain free variables.

To prove that \( \bigcup F'^{i} \) is regular, we distinguish between \( t = a \) and \( t \neq a \). The former case is trivial. If \( t \neq a \), then the equation \( x = t[a := x] \) has a unique, regular, solution (property 8). But since \( F \) is continuous (see [Cou83], proposition 3.3.3), \( \bigcup F'^{i} = F(\bigcup F'^{i}) = t[a := \bigcup F'^{i}] \), that solution must be \( \bigcup F'^{i} \).

\( \square \)
the model for the terms

To complete the model, we have to construct a family of cpos $Dom$ that solves the system of coupled domain equations:

\[ \forall a \in Kind_* : \quad Dom_a \cong F_a(Dom) \]

i.e.

\[
\begin{align*}
Dom[\sigma] & = Dom \sigma \\
Dom[\sigma \rightarrow \tau] & = \mathbb{P}(Dom) \cong [Dom[\sigma] \rightarrow Dom[\tau]] \\
Dom[\Pi^\omega] & = \prod_{\sigma \in Kind_*} Dom[\sigma]^\omega \\
Dom[\mu(\lambda a \cdot a)] & = Dom_1 \cong Dom_1
\end{align*}
\]

We define the functor $F : \mathcal{K} \rightarrow \mathcal{K}$ by $F = \{ F_a \mid a \in Kind_* \}$, where the functors $F_a : \mathcal{K} \rightarrow \mathbb{CPO}_{PR}$ are defined by

\[
\begin{align*}
F_\sigma & = C_{\text{domain}_\sigma} \\
F & = \mathbb{P}\mathbb{R} \times \mathbb{P}_\sigma \\
F_\Pi^\omega & = \mathbb{P}\mathbb{P}_\sigma \times \mathbb{P}_{\tau[\sigma = a]} \mid a \in Kind_* \\
F_1 & = \bot
\end{align*}
\]

The initial fixed point of $F$ gives the cpos $Dom_a$ satisfying the recursive domain equations, and the associated isomorphisms $\Phi_a \in Dom_a \rightarrow F_a(Dom)$. Again, the cpo $Dom[\mu(\lambda a \cdot a)]$ will be the one-point cpo.
4 Subtyping

We now consider the extension of system \( A \) with subtyping. This system will be called \( A \subseteq \).

4.1 Syntax

We will have a subtype relation \( \leq \) on types. If \( \sigma \leq \tau \), we say that \( \sigma \) is a subtype of \( \tau \). The subtype relation will be a pre-order (i.e., reflexive and transitive).

We add the following type inference rule: the subsumption rule

\[
\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash \sigma \leq \tau}{\Gamma \vdash M : \tau} (\text{SUB})
\]

This means that terms no longer have a unique type.

The subtype relation will be based on a subtype relation \( \leq^B \) on the base types. For example, if \( \text{int} \) and \( \text{real} \) are base types, we could have \( \text{int} \leq^B \text{real} \).

We have the following rules for deducing \( \sigma \leq \tau \) for more complex types \( \sigma \) and \( \tau \).

\[
\frac{\sigma \leq^B \tau}{\Gamma \vdash \sigma \leq \tau} \quad \frac{\Gamma \vdash \sigma =_e \tau}{\Gamma \vdash \sigma \leq \tau} (\text{REFL}) \quad \frac{\Gamma \vdash \rho \leq \sigma \quad \Gamma \vdash \sigma \leq \tau}{\Gamma \vdash \rho \leq \tau} (\text{TRANS})
\]

\[
\frac{\Gamma \vdash \sigma \leq \tau \quad \Gamma \vdash \tau \leq \tau'}{\Gamma \vdash \sigma \leq \tau'} (\leq -) \quad \frac{\Gamma, \alpha : * \vdash f \alpha \leq g \alpha}{\Gamma, \Pi f \leq \Pi g} (\leq II)
\]

Note the contravariance of \( \_ \) with respect to the subtype relation.

That \( \leq \) is indeed a pre-order is of course guaranteed by the rule (REFL) and (TRANS). The rule (TEQ) is omitted, since it can be derived from (REFL) and (SUB).

For the model construction we will need the following lemma.

14 lemma \( \Gamma \vdash \sigma \rightarrow \tau \leq \sigma' \rightarrow \tau' \Rightarrow \Gamma \vdash \sigma \leq \sigma' \) and \( \Gamma \vdash \tau \leq \tau' \)

\( \Gamma \vdash \Pi f \leq \Pi g \Rightarrow \Gamma, \alpha : * \vdash f \alpha \leq g \alpha \)

This lemma can be proved as follows. We define a relation \( \leq' \) on types. For \( \leq' \) we have the same derivation rules as for \( \leq \), except instead of (TRANS) we have the following rule

\[
\frac{\Gamma \vdash \sigma \leq' \tau \quad \Gamma \vdash \sigma =_e \sigma' \quad \Gamma \vdash \tau =_e \tau'}{\Gamma \vdash \sigma \leq' \tau'} (\text{LEQ})
\]

Clearly \( \Gamma \vdash \sigma \leq' \tau \Rightarrow \Gamma \vdash \sigma \leq \tau \).

By the next lemma we also have \( \Gamma \vdash \sigma \leq \tau \Rightarrow \Gamma \vdash \sigma \leq' \tau \).

15 lemma \( \leq' \) is transitive, i.e., \( \Gamma \vdash \rho \leq' \sigma \quad \Gamma \vdash \sigma \leq' \tau \Rightarrow \Gamma \vdash \rho \leq' \tau \)

proof By induction on the derivation length, not counting the rule (TEQ).

Suppose \( \Gamma \vdash \rho \leq' \sigma \) and \( \Gamma \vdash \sigma \leq' \tau \). Then

(a) \( \rho, \sigma \) and \( \tau \) are \( \beta\eta \)-equal to base types \( \alpha, \beta \) and \( \gamma \), respectively, and \( \alpha \leq^B \beta \leq^B \gamma \), or

(b) \( \rho =_e \rho_1 \rightarrow \rho_2, \sigma =_e \sigma_1 \rightarrow \sigma_2 \) and \( \tau =_e \tau_1 \rightarrow \tau_2 \), or

(c) \( \rho =_e \Pi f, \sigma =_e \Pi g \) and \( \tau =_e \Pi h \).
We must prove $\Gamma \vdash \rho \leq^{'} \tau$. For (a) this is trivial. We will give the proof for (b). The proof for (c) is similar.

The derivations of $\Gamma \vdash \rho \leq^{'} \sigma$ and $\Gamma \vdash \sigma \leq^{'} \tau$, must both end with $(\leq \text{EQ})$, possibly followed by $(\leq \text{TEQ})$. So $\Gamma \vdash \sigma_1 \leq \rho_1$, $\Gamma \vdash \rho_2 \leq^{'} \sigma_2$, $\Gamma \vdash \tau_1 \leq \sigma_1$ and $\Gamma \vdash \sigma_2 \leq \tau_2$.

By the induction hypothesis $\Gamma \vdash \tau_1 \leq \rho_1$ and $\Gamma \vdash \rho_2 \leq^{'} \tau_2$, and hence $\Gamma \vdash \rho \leq^{'} \tau$

$\square$

So $\Gamma \vdash \sigma \leq^{'} \tau \iff \Gamma \vdash \sigma \leq \tau$, and for $\leq^{'}$ it is obvious that lemma 14 holds. In fact, we already used it in the proof of lemma 15.

4.2 Semantics: general model definition

Because the semantics of terms is defined by induction on type derivations, we have to define the semantics of the new type inference rule, the subsumption rule.

Suppose $\Gamma \vdash M : \tau$ is derived from from $\Gamma \vdash M : \sigma$ and $\Gamma \vdash \sigma \leq \tau$:

$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash \sigma \leq \tau}{\Gamma \vdash M : \tau} \text{(SUB)}$

Since $[\Gamma \vdash M : \sigma] \eta \in \text{Dom}_{[\Gamma \vdash \sigma : \eta]}$, and we want $[\Gamma \vdash M : \tau] \eta \in \text{Dom}_{[\Gamma \vdash \tau : \eta]}$, we need a coercion function from $\text{Dom}_{[\Gamma \vdash \sigma : \eta]}$ to $\text{Dom}_{[\Gamma \vdash \tau : \eta]}$. We will call this function $\text{Coe}_{[\Gamma \vdash \sigma : \eta]} [\Gamma \vdash \tau : \eta]$.

We can now give the meaning of $M : \tau$ in terms of the meaning of $M : \sigma$

$[\Gamma \vdash M : \tau] \eta = \text{Coe}_{[\Gamma \vdash \sigma : \eta]} [\Gamma \vdash \tau : \eta] [\Gamma \vdash M : \sigma] \eta$

For all types $\sigma$ and $\tau$ such that $\Gamma \vdash \sigma \leq \tau$, we need a coercion function from $\text{Dom}_{[\Gamma \vdash \sigma : \eta]}$ to $\text{Dom}_{[\Gamma \vdash \tau : \eta]}$. We require that the coercion functions are continuous.

Not any set of coercion function will do. Remember that the meaning of a term is defined by induction on its type derivation. Not only will there be more than one type derivation for $\Gamma \vdash M : \sigma$, but in different derivations a subexpression of $M$ may have different types and hence different meanings. We have to prove coherence, that all derivations for $\Gamma \vdash M : \sigma$ give the same meaning $[\Gamma \vdash M : \sigma] \eta$.

We will now try to find some additional requirements for the coercion functions to guarantee that an environment model is coherent.

**notation** The same conventions we use to abbreviate the subscripts of the form Dom will be used for the subscripts of Coe and $\Phi$. So we will write $\text{Coe}_{[\sigma]} [\tau]$ instead of $\text{Coe}_{[\Gamma \vdash \sigma : \eta]} [\Gamma \vdash \tau : \eta]$ and $\Phi_{[\sigma]} [\tau]$ instead of $\Phi_{[\Gamma \vdash \sigma : \eta]} [\Gamma \vdash \tau : \eta]$. When we are dealing with the term model for the constructor expressions, we will just write $\text{Coe}_{\sigma}$ and $\Phi_{\sigma}$.

**coherence**

The subsumption rule itself gives rise to the following two fairly obvious requirements for the coercion functions:

$P_0 \quad \text{Coe}_{a} \xi = \lambda \xi \in \text{Dom}_{a} \xi \quad \text{for all } a \in \text{Kind}.$

$P_1 \quad \text{Coe}_{a \leq b} = \text{Coe}_{b} \circ \text{Coe}_{a} \quad \text{for all } a \leq b \leq c$
16 lemma If $\mathcal{P}_0$ or $\mathcal{P}_1$ does not hold, then the semantics is not coherent. \\
proof The subtype relation is reflexive, so

$$\Gamma \vdash M : \sigma \quad \Gamma \vdash \sigma \leq \tau$$

which yields

$$[\Gamma \vdash M : \sigma] \eta = \text{Coe}_{\eta} \llbracket \sigma \rrbracket \eta \quad \eta$$

So if $\mathcal{P}_0$ does not hold, then $\Gamma \vdash M : \sigma$ does not have a unique meaning. \\
Suppose $\Gamma \vdash \rho \leq \sigma \leq \tau$. Then

$$\Gamma \vdash M : \rho \quad \Gamma \vdash \rho \leq \tau$$

yields $[\Gamma \vdash M : \tau] \eta = \text{Coe}_{\rho} \llbracket \tau \rrbracket (\text{Coe}_{\rho} \llbracket \sigma \rrbracket [\Gamma \vdash M : \rho] \eta)$

but

$$\Gamma \vdash M : \rho \quad \Gamma \vdash \rho \leq \sigma \quad \Gamma \vdash \sigma \leq \tau$$

yields $[\Gamma \vdash M : \tau] \eta = \text{Coe}_{\rho} \llbracket \tau \rrbracket (\text{Coe}_{\rho} \llbracket \sigma \rrbracket [\Gamma \vdash M : \rho] \eta)$

So if $\mathcal{P}_1$ does not hold, then $\Gamma \vdash M : \tau$ does not have a unique meaning. □

$\mathcal{P}_0$ and $\mathcal{P}_1$ are not sufficient to have coherence. We will also require properties of the coercions between $\rightarrow$-types and $\Pi$-types.

First we consider function types. Let $\sigma' \leq \sigma$ and $\tau \leq \tau'$, so $\sigma \rightarrow \tau \leq \sigma' \rightarrow \tau'$. Suppose $\Gamma \vdash M : \sigma \rightarrow \tau$ and $\Gamma \vdash N : \sigma'$. Then $\Gamma \vdash MN : \tau'$ can be derived in several ways, for instance:

(i) $\quad \frac{M : \sigma \rightarrow \tau \quad \sigma \rightarrow \tau \leq \sigma' \rightarrow \tau'}{N : \sigma'}$ \quad (ii) $\quad \frac{M : \sigma \rightarrow \tau}{\frac{\Gamma \vdash M : \sigma \rightarrow \tau}{\frac{\Gamma \vdash N : \sigma' \quad \sigma' \leq \sigma}{\lambda N : \sigma'}}}$

These two derivations give as $[\Gamma \vdash MN : \tau'] \eta$

$$\Phi (\text{Coe}_{\sigma \rightarrow \tau} \llbracket \sigma' \rightarrow \tau' \rrbracket (\text{Coe}_{\sigma} \llbracket \tau \rrbracket)) \quad (\text{Coe}_{\sigma} \llbracket \sigma' \rrbracket (\text{Coe}_{\sigma} \llbracket \tau \rrbracket))$$

In order for these to be equal, some equation between $\text{Coe}_{\sigma \rightarrow \tau} \llbracket \sigma' \rightarrow \tau' \rrbracket$ and $\text{Coe}_{\sigma} \llbracket \sigma' \rrbracket$ and $\text{Coe}_{\sigma} \llbracket \tau \rrbracket$ has to hold. There is really only one way to express a relation between $\text{Coe}_{\sigma \rightarrow \tau} \llbracket \sigma' \rightarrow \tau' \rrbracket$ and $\text{Coe}_{\sigma} \llbracket \sigma' \rrbracket$ and $\text{Coe}_{\sigma} \llbracket \tau \rrbracket$.

$\mathcal{P}_2$ : for all $\Gamma \vdash \rho' \leq \sigma$ and $\Gamma \vdash \tau \leq \tau'$

$$\text{Coe}_{\sigma \rightarrow \tau} \llbracket \rho' \rightarrow \tau \rrbracket = \Phi^{-1}_{\sigma \rightarrow \tau} \circ FS(\text{Coe}_{\sigma} \llbracket \rho' \rrbracket, \text{Coe}_{\tau} \llbracket \tau' \rrbracket) \circ \Phi_{\sigma \rightarrow \tau}$$

If $\mathcal{P}_2$ holds, then (i) and (ii) give the same meaning for $\Gamma \vdash MN : \tau'$.
Now we consider polymorphic types. Let \( r, a : * \vdash f \alpha \leq g \alpha \) so \( \Pi f \leq \Pi g \), \( \Gamma \vdash \sigma : * \) and \( \Gamma \vdash M : \Pi f \).
Then \( f \sigma \leq g \sigma \), and \( \Gamma \vdash M \sigma : g \sigma \) can be derived in several ways, for example:

\[
\frac{M : \Pi f \quad \Pi f \leq \Pi g}{M : \Pi g} \quad \sigma : * \quad \frac{M : \Pi f \quad \sigma : *}{M : \Pi g} \quad \frac{M : \Pi f \quad \sigma : *}{f \sigma \leq g \sigma}
\]

These two derivations give for \( [\Gamma \vdash M \sigma : g \sigma] \eta \)

\[
\Phi(\text{Coec}_{[\Pi f]} \Pi g) \quad [\Gamma \vdash M : \Pi f] \eta) \quad [\Gamma \vdash \sigma : *] \eta
\]

(ii) \( \text{Coec}_{[\sigma]} ([\Phi(\Gamma \vdash M : \Pi f] \eta) \quad [\Gamma \vdash \sigma : *] \eta) \)

Of course, we want these to be equal.

Again, there is only one way we can express a relation between \( \text{Coec}_{[\Pi f]} \Pi g \) and \( \text{Coec}_{[\sigma]} \).

\[
\text{Dom}_{[\Pi f]} \quad \cong \quad \Pi_{[\alpha] \in \text{Kind}_a} \text{Dom}_{[\alpha]}
\]

\[
\text{Dom}_{[\Pi f]} \quad \cong \quad \Pi_{[\alpha] \in \text{Kind}_a} \text{Dom}_{[\alpha]}
\]

\[
P_3 : \text{for all } a, \alpha : * \vdash f \alpha \leq g \alpha
\]

\[
\text{Coec}_{[\Pi f]} \Pi g = \Phi_{[\Pi f]}^{-1} GP (< \text{Coec}_{[\Pi f]} \Pi g) \quad [\alpha] \in \text{Kind}_a > \phi_{[\Pi f]}
\]

If \( P_3 \) holds, then (i) and (ii) do indeed give the same meaning for \( \Gamma \vdash M \sigma : g \sigma \).

So we now have the following requirements for the coercion functions

\[
P_0 \quad \text{for all } a \in \text{Kind}_a
\]

\[
\text{Coec}_a a
\]

\[
P_1 \quad \text{for all } a \leq^* b \leq^* c
\]

\[
\text{Coec}_a a = \text{Coec}_b b
\]

\[
P_2 \quad \text{for all } \Gamma \vdash \sigma \leq \sigma' \text{ and } \Gamma \vdash \tau \leq \tau'
\]

\[
\text{Coec}_{[\sigma \vdash \tau]} ([\sigma' \vdash \tau']) = \Phi_{[\sigma' \vdash \tau']}^{-1} \epsilon FS(\text{Coec}_{[\sigma]} \Pi \sigma, \text{Coec}_{[\tau]} \Pi \tau) \phi_{[\sigma \vdash \tau]}
\]

\[
P_3 \quad \text{for all } \Gamma, \alpha : * \vdash f \alpha \leq g \alpha
\]

\[
\text{Coec}_{[\Pi f]} \Pi g = \Phi_{[\Pi f]}^{-1} GP (< \text{Coec}_{[\Pi f]} \Pi g) \quad [\alpha] \in \text{Kind}_a > \phi_{[\Pi f]}
\]

If the coherence conditions \( P_0, P_1, P_2 \) and \( P_3 \) hold, then the semantics is coherent. In fact, the semantics is coherent if and only if these conditions are satisfied. The proof can be found in the appendix. For the proof we use the fact that we have minimal typing in \( A_{\leq} \).

The subtype relation \( \leq \) on types induces a subtype relation \( \leq^* \) on \( \text{Kind}_a \).

\[
a \leq^* b
\]

iff

\[
\exists \Gamma, \eta, \sigma, \tau, \Gamma \vdash \eta \quad [\Gamma \vdash \sigma : *] \eta = a \quad [\Gamma \vdash \tau : *] \eta = b \quad \Gamma \vdash \sigma \leq \tau
\]

which is the same as

\[
[\Gamma \vdash \sigma : *] \eta \leq^* [\Gamma \vdash \tau : *] \eta
\]

iff

\[
\Gamma \vdash \sigma \leq \tau
\]

Because \( \leq \) is a pre-order, so is \( \leq^* \). Once we have decided on a particular submodel for the constructors, we will give a simpler and more workable definition of \( \leq^* \).

So we get the following model definition for \( A_{\leq} \)

28
17 definition (general model definition $\langle \leq \rangle$)
An environment model for $\langle \leq \rangle$ is a 7-tuple
$< \text{Kind}, \Phi_{\text{cont}}, \mathcal{I}_{\text{cont}}, \text{Dom}, \Phi_{\text{term}}, \mathcal{I}_{\text{term}}, \text{Coe} >$,
where Coe is a family of coercion functions satisfying $\mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2$ and $\mathcal{P}_3$.

$$\text{Coe} = \langle \text{Coe}_{a b} | \text{for all } a, b \in \text{Kind}, a \leq^* b \rangle$$

where for all $a \leq^* b$

$$\text{Coe}_{a b} \in [\text{Dom}_a \rightarrow \text{Dom}_b]$$

and the rest as in definition 4, with the definition of $\leq^*$ for the subsumption rule given by

$$\Gamma \vdash M : \tau \eta = \text{Coe}_{\Gamma \vdash \sigma : *}[\Gamma \vdash \tau : *]_\eta \Gamma \vdash M : \sigma \eta$$

\(\Box\)

4.3 The construction of a cpo model

We will use the same submodel for the constructor expressions we used for $A$, i.e. a term model.
Because we have a term model we can define $\leq^*$ as follows:

18 definition ($\leq^*$)
If $a, b \in \text{Kind}$, then $a$ and $b$ are closed type expressions, i.e. $\triangleright \vdash a : *$ and $\triangleright \vdash b : *$, and so we can define $\leq^*$ by

$$a \leq^* b \text{ iff } \triangleright \vdash a \leq b$$

\(\Box\)

19 lemma $\Gamma \vdash \sigma \leq \tau \iff \forall \eta [\Gamma \vdash \sigma : * \eta \leq^* [\Gamma \vdash \tau : *] \eta]$

proof By induction on $\sigma$ or $\tau$. \(\Box\)

Before we can begin to construct a cpo-model for $\langle \leq \rangle$, some coercions have to be given. We need coercions
functions $\text{coerce}_{\sigma \tau}$ from $\text{domain}_\sigma$ to $\text{domain}_\tau$, for all base types $\sigma$ and $\tau$ such that $\sigma \leq^B \tau$. We require that these coercion functions are continuous, and that $\mathcal{P}_0$ and $\mathcal{P}_1$ hold, ie.

$$\text{coerce}_{\sigma \tau} = \lambda \xi \in \text{domain}_\sigma. \xi$$

$$\text{coerce}_{\rho \tau} = \text{coerce}_{\rho \tau} \circ \text{coerce}_{\rho \sigma} \text{ if } \rho \leq^B \sigma \leq^B \tau$$

For $\sigma \leq^B \tau$, $\text{Coe}_{\sigma \tau} \in [\text{Dom}_\sigma \rightarrow \text{Dom}_\tau]$ is of course defined by

$$\text{Coe}_{\sigma \tau} = \Phi^{-1}_\tau \circ \text{coerce}_{\sigma \tau} \circ \Phi_\sigma$$

So we are looking for a family of cpos $\langle \text{Dom}_a | a \in \text{Kind} \rangle$, solving the coupled domain equations

$$\text{Dom}_\sigma \cong \text{domain}_\sigma$$

$$\text{Dom}_{\sigma \rightarrow \tau} \cong \text{FS}(<\text{Dom}_\sigma, \text{Dom}_\tau>)$$

$$\text{Dom}_{\text{if}} \cong \text{GP}(<\text{Dom}_{f a} | a \in \text{Kind}>)$$
and a family of coercion functions $\langle \text{Coe}_a \rangle_{a \leq^* b}$ satisfying $P_0$, $P_1$ and

\[
\begin{align*}
\text{Coe}_\sigma \tau & = \Phi_{\tau\sigma}^{-1} \circ \text{coerce}_\sigma \tau \circ \Phi_{\sigma} \\
\text{Coe}_\sigma \rightarrow_{\sigma'} \tau \rightarrow_{\tau'} & = \Phi_{\tau',\tau}^{-1} \circ FS(\text{Coe}_{\sigma'} \sigma, \text{Coe}_{\tau'} \tau) \circ \Phi_{\sigma \rightarrow \tau} \\
\text{Coe} \Pi_f \Pi_g & = \Phi_{\Pi_f,\Pi_g}^{-1} \circ GP(\langle \text{Coe}_{f \circ g} \rangle_{a \in \text{Kind.} \setminus \rightarrow} \rangle) \circ \Phi_{\Pi_f} \quad \text{for all } \Pi_f \leq^* \Pi_g
\end{align*}
\]

We define the category corresponding with the subtype relation on $\text{Kind.}$.

20 definition $(\text{Kind}.)$

The objects of $\text{Kind.}$ are the elements of $\text{Kind.}$, and there is a unique arrow, called $a \leq b$, from $a$ to $b$ iff $a \leq^* b$.

Because $\leq^*$ is reflexive, there is an identity $a : a$ for all objects $a$. Because $\leq^*$ is transitive, composition is always defined: $b \leq a \leq c$ will be $b \leq c$.

Together, $\text{Dom}$ and $\text{Coe}$ can be seen as a functor from $\text{Kind.}$ to $\text{CPO}$. $\text{Dom}$ is the object part, mapping every $\text{Kind.}$-object, i.e., every element of $\text{Kind.}$, to a $\text{CPO}$-object, a cpo. $\text{Coe}$ is the morphism part, mapping every $\text{Kind.}$-morphism $ab$ to a continuous function from $\text{Dom}a$ to $\text{Dom}b$.

For this to be a functor, identities and composition must be preserved. This is guaranteed by $P_0$ and $P_1$.

$\langle \text{domain} \rangle_{\sigma}$ a base type and $\langle \text{coerce} \rangle_{\sigma}$ $\sigma \leq^B \tau$ form a functor from the category corresponding with the pre-order $\leq^B$ on base types to $\text{CPO}$.

We will construct $\text{Dom} \& \text{Coe}$, the functor formed by $\text{Dom}$ and $\text{Coe}$ together, as an initial fixed point in a functor category. Because of the contravariance of $FS$ in its first argument, we cannot construct $\text{Dom}$ in the standard functor category $[\text{Kind.}, \text{CPO}]$ (usually written $\text{CPO}^{\text{Kind.}}$). Instead, we work in the associated category of embedding-projection pairs. Morphisms of $[\text{Kind.}, \text{CPO}]$ are natural transformations, families of $\text{CPO}$-morphisms. So pointwise, they have the same properties as $\text{CPO}$-morphisms, in particular those properties that enable the use of embedding-projection pairs.

$\text{CPO}_1$ is the category with cpos as objects and strict continuous functions as morphisms. It is a subcategory of $\text{CPO}$.

21 definition $[\text{Kind.}, \text{CPO}_1]_{PR}$

$[\text{Kind.}, \text{CPO}_1]_{PR}$ is the category with as objects functors from $\text{Kind.}$ to $\text{CPO}_1$, and as morphisms embedding-projection pairs of natural transformations: if $F$ and $G$ are functors from $\text{Kind.}$ to $\text{CPO}_1$, then $(\eta, \theta)$ is a morphism from $F$ to $G$ if

\[
\begin{align*}
\eta & : F \rightarrow G \\
\theta & : G \rightarrow F
\end{align*}
\]

and for all $a \in \text{Kind.}$, $\theta_a \circ \eta_a = \text{id}_F \circ a$

\[
\eta_a \circ \theta_a = \text{id}_G \circ a
\]

Composition is of course defined by $(\eta, \theta) \circ (\eta', \theta') = (\eta' \circ \eta, \theta \circ \theta')$.

$\square$
The reason for using $CPO_\perp$ instead of $CPO$, is that $[Kind, CPO]_{PR}$ is not an $\omega$-category, because it does not have an initial element.

**22 lemma** $[Kind, CPO_\perp]_{PR}$ is an $\omega$-category

**proof** see [Pol91] □

As a consequence of using $CPO_\perp$ instead of $CPO$, the coercion functions $\text{coerce}_{\sigma\tau}$ will have to be strict. ¹ Because $CPO_\perp$ is a subcategory of $CPO$ and $FS$ and $GP$ preserve strictness, $FS : CPO\perp^{OP} \times CPO\perp \to CPO\perp$ and $GP : \Pi CPO\perp \to CPO\perp$.

$\text{Dom}&\text{Coe}$ will be the initial fixed point of the following functor $\mathbf{F}$

**23 definition** $(\mathbf{F} : \mathcal{K} \to \mathcal{K})$

$\mathbf{F}$ is a functor $\mathcal{K}$ to $\mathcal{K}$, so it consists of an object part, a mapping from $\text{Obj}(\mathcal{K})$ to $\text{Obj}(\mathcal{K})$, and an morphism part, a mapping from $\text{Mor}(\mathcal{K})$ to $\text{Mor}(\mathcal{K})$.

The object part of $\mathbf{F}$ is defined as follows. Let $F \in \text{Obj}(\mathcal{K})$. Then $\mathbf{F} F \in \text{Obj}(\mathcal{K})$, ie. $\mathbf{F} F$ is a functor from $\text{Kind}^{\perp}$ to $CPO_\perp$.
The object part of $\mathbf{F} F$, a mapping from $\text{Obj}(\text{Kind}^{\perp})$ to $\text{Obj}(CPO_\perp)$, is defined by

\[
\begin{align*}
(\mathbf{F} F)a &= \text{domain}_{a} \\
(\mathbf{F} F)a \to b &= FS(Fa, Fb) \\
(\mathbf{F} F)\Pi f &= GP(< F(fa) | a \in \text{Kind}^{\perp} >)
\end{align*}
\]

and the morphism part of $\mathbf{F} F$, a mapping from $\text{Mor}(\text{Kind}^{\perp})$ to $\text{Mor}(CPO_\perp)$, is defined by

\[
\begin{align*}
(\mathbf{F} F)a \leq b &= \text{coerce}_{ab} \\
(\mathbf{F} F)a \to b \leq a' \to b' &= FS(Fa' \leq a, Fb \leq b') \\
(\mathbf{F} F)\Pi f \leq \Pi g &= GP(< Ffa \leq ga | a \in \text{Kind}^{\perp} >)
\end{align*}
\]

The morphism part of $\mathbf{F}$ is defined as follows:

if $(\eta, \theta) \in \text{Hom}_{\mathcal{K}}(F, G)$, so $\eta : F \sigma \to G$ and $\theta : G \tau \to F$ then

$\mathbf{F}(\eta, \theta) = (\eta', \theta')$, ie. $\eta' : \mathbf{F} F \sigma \to \mathbf{F} G$ and $\theta' : \mathbf{F} G \tau \to \mathbf{F} F$ where

\[
\begin{align*}
(\eta'', \theta'') &= (\text{id}_{\text{domain}_{a}}, \text{id}_{\text{domain}_{a}}) \\
(\eta'_{a}, \theta''_{a}) &= FSPR((\eta_{a}, \theta_{a}), (\eta_{b}, \theta_{b})) \\
(\eta''_{a}, \theta''_{a}) &= GPSPR(< (\eta_{a}, \theta_{a}) | a \in \text{Kind}^{\perp} >)
\end{align*}
\]

Checking $\eta' : \mathbf{F} F \sigma \to \mathbf{F} G$ and $\theta' : \mathbf{F} G \tau \to \mathbf{F} F$ is straightforward, and it can easily be verified (pointwise) that $\mathbf{F}$ preserves identities and composition.

□

Note that for the coercions $FS$ is used, which takes care of the contravariance of $\to$ with respect to the subtype relation

$\Gamma \vdash \sigma' \leq \sigma \quad \Gamma \vdash \tau \leq \tau'$

$\Gamma \vdash \sigma \to \tau \leq \sigma' \to \tau'$

whereas for the morphisms $FS_{PR}$ is used, which is covariant in both arguments, so that a fixed point can be constructed.

Any fixed point of $\mathbf{F}$ will solve the recursive domain equations and satisfy the conditions for the coercion functions.

¹The requirement that the coercions be strict also comes up in [BTCGS89], although for different reasons.
For instance, let \((F, (\Phi, \Psi))\) be a fixed point of \(\mathcal{F}\), i.e. \((\Phi, \Psi)\) is an isomorphism between \(F\) and \(\mathcal{F}F\). This means that \(\Phi : F \rightarrow \mathcal{F}F\) and \(\Psi : \mathcal{F}F \rightarrow F\), such that \(\Phi \circ \Psi = id_{\mathcal{F}F}\) and \(\Psi \circ \Phi = id_F\).

Because everything is defined pointwise, this means that for all \(a \leq b\)

\[
\begin{align*}
\Phi_b \circ \Psi_b &= id_{(\mathcal{F}F)b} \quad \text{and} \\
\Psi_b \circ \Phi_b &= id_{\mathcal{F}F} \\
\Phi_a \circ \Psi_a &= id_{(\mathcal{F}F)a} \\
\Psi_a \circ \Phi_a &= id_{\mathcal{F}Fa}
\end{align*}
\]

Let \(\Pi f \leq \Pi g\). Then

\[
\begin{align*}
\Phi_{\Pi f} &\quad (F \Pi f = GP(< F(fa) | a \in Kind_*>) \\
F\Pi f \leq \Pi g &\quad (F \Pi f \leq \Pi g = GP(< F(fa)(ga) | a \in Kind_*>) \\
\Phi_{\Pi g} &\quad (F \Pi g = GP(< F(ga) | a \in Kind_*>))
\end{align*}
\]

and \(F \Pi g \leq \Pi f = \Psi_{\Pi g} \circ ((F \Pi f) \Pi f \leq \Pi g) \circ \Phi_{\Pi f} = \Psi_{\Pi g} \circ GP(< F(fa) \leq ga | a \in Kind_*>) \circ \Phi_{\Pi f}\)

so \(P_3\) is satisfied. In the same way it can be shown that \(P_2\) is satisfied.

**24 lemma** \(\mathcal{F}\) is \(\omega\)-continuous

**proof** (sketch, for details see [Pol91])

We define a functor \(\mathcal{H} : [\text{Kind}_*, \text{CPO}]^{op} \times [\text{Kind}_*, \text{CPO}] \rightarrow [\text{Kind}_*, \text{CPO}]\) such that

\[
\begin{align*}
\mathcal{F}F &= \mathcal{H}(F,F) \\
\mathcal{F}(\eta, \theta) &= (\mathcal{H}(\eta, \theta), \mathcal{H}(\eta, \theta)) = \mathcal{H}_{PR}(\eta, \theta), (\eta, \theta))
\end{align*}
\]

We prove so-called local continuity for \(\mathcal{H}\), which can be done pointwise. This means that \(\mathcal{H}_{PR}\) is \(\omega\)-continuous. Using the correspondence between \(\mathcal{F}\) and \(\mathcal{H}_{PR}\) given above, we can prove that if \(\mathcal{H}_{PR}\) is \(\omega\)-continuous, \(\mathcal{F}\) is also \(\omega\)-continuous.

So by the initial fixed point theorem \(\mathcal{F}\) has an initial solution \((\text{Dom} \& \text{Coe}, (\Phi, \Psi))\). The object part of \(\text{Dom} \& \text{Coe} \mathcal{F}\) gives us the family of cpos \(\text{Dom}\), the morphism part gives us the family of coercions \(\text{Coe}\), and \(\Phi\) is the required family of bijections.

So, recapitulating,

- \(\text{CPO}_1\) is an O-category
- \([\text{Kind}_*, \text{CPO}_1]\) is an O-category
- \([\text{Kind}_*, \text{CPO}_1]\) is an \(\omega\)-category
• $\mathcal{F}$ is $\omega$-continuous

• in $[\text{Kind}, \ CPO]\_{PR}$ the equation $\mathcal{F}(D) \cong D$ has an initial solution $(\text{Dom}&\text{Coe}, (\Phi, \Psi))$

• $(\text{Dom}&\text{Coe}, (\Phi, \Psi))$ gives us a family of cpos solving the recursive domain equations with the associated bijections, and a family of coercions satisfying the coherence conditions.
5 Recursive types and subtyping

We will now combine the two extensions of $\Lambda$ we have dealt with, subtyping and recursive types.

5.1 Syntax

First we consider how to define the subtype relation on recursive types. The natural rule for subtyping on recursive types is

$$\Gamma, \alpha : *, \beta : *, \alpha \leq \beta \vdash f\alpha \leq g\beta$$

This is the same as

$$\Gamma \vdash \mu f \leq \mu g$$

where $\alpha$ may only occurs at covariant positions in $f\alpha$ or $g\alpha$.

Contexts can now also also contain expressions of the form $\alpha \leq \beta$, where $\alpha$ and $\beta$ are type variables, but only when we are deriving subtype judgements. In $\Gamma \vdash M : \sigma$ the context will not contain expressions of the form $\alpha \leq \beta$.

We will now also need the rule

$$\Gamma, \alpha \leq \beta \vdash \alpha \leq \beta$$

Since we considered three ways to incorporate recursive types in $\Lambda$, several options are open to us. The systems we get by extending $\Lambda\mu_1, \Lambda\mu_2$ and $\Lambda\mu_3$ with subtyping will be called $\Lambda_{\leq}\mu_1, \Lambda_{\leq}\mu_2$ and $\Lambda_{\leq}\mu_3$, respectively.

Since in $\Lambda\mu_i \mu f \neq f(\mu f)$, we could add the following rules for $\Lambda_{\leq}\mu_i$,

$$\Gamma \vdash f : * \Rightarrow * \quad \Gamma \vdash f : * \Rightarrow *$$

The $fold_{\mu_f}$ and $unfold_{\mu_f}$ can then be omitted. The coercions for $\mu f \leq f(\mu f)$ and $f(\mu f) \leq \mu f$ are of course $\Phi_{\mu_f}$ and $\Phi_{\mu_f}^{-1}$. However, the resulting system is then virtually the same as $\Lambda_{\leq}\mu_2$, because the same type derivations $\Gamma \vdash M : \sigma$ will be derivable. The only difference is the notion of constructor equality.

5.2 Semantics: general model definition

Remember that we can now have expressions such as $\alpha \leq \beta$ in contexts. For an environment $\eta$ to satisfy a context $\Gamma$ we now also require that

$$\eta(\alpha) \leq^* \eta(\beta) \quad \text{for all } (\alpha \leq \beta) \in \Gamma$$

We get environment models for these systems with subtyping and recursive types by extending a model for the corresponding system without subtyping with a family of coercion functions $Coe =< Coe_{a,b} | a, b \in \text{Kind}, a \leq^* b >$

25 definition (general model definition $\Lambda_{\leq}\mu_1, \Lambda_{\leq}\mu_2$ and $\Lambda_{\leq}\mu_3$)

A second order environment model for $\Lambda_{\leq}\mu_1, \Lambda_{\leq}\mu_2$ or $\Lambda_{\leq}\mu_3$ is a 7-tuple

$$< \text{Kind}, \Phi_{\text{cons}}, I_{\text{cons}}, \text{Dom}, \Phi_{\text{term}}, I_{\text{term}}, Coe >$$

where $Coe$ is a family of coercion functions,

$$Coe =< Coe_{a,b} \in [\text{Dom}_a \rightarrow \text{Dom}_b] | a, b \in \text{Kind}, a \leq^* b >$$
satisfying \( P_0, P_1, \) Parrow and \( P_3, \) and the rest as in the definition of the general model definition for \( \Lambda \mu_1, \Lambda \mu_2 \) or \( \Lambda \mu_3 \) (definitions 5, 6 and 10).

\( \Box \)

26 theorem (coherence)
The semantics of \( \Lambda \leq \mu_1, \Lambda \leq \mu_2 \) and \( \Lambda \leq \mu_3 \) are coherent
proof
For the systems \( \Lambda \leq \mu_2 \) and \( \Lambda \leq \mu_3 \) we have the same type inference rules as for \( \Lambda \leq \). So the proof of coherence for \( \Lambda \leq \) (theorem 39) also proves coherence for \( \Lambda \leq \mu_2 \) and \( \Lambda \leq \mu_3 \).
The two extra type inference rules that we have in \( \Lambda \leq \mu_3 \) viz. \( (FOLD) \) and \( (UNFOLD) \)
do not pose a problem as far as coherence is concerned, because of the subscripts of \( fold_{\mu f} \) and \( unfold_{\mu f} \)
\( \Box \)

For the model constructions we only have to define a pre-order on \( \text{Kind} \) that corresponds with the subtype relation on types. We can then construct a model in the same way as we did for \( \Lambda \leq \), as an initial fixed point of a functor \( \mathcal{F} \) on \( \text{PPO}_{\text{Kind}} \).

5.3 The construction of a cpo model for \( \Lambda \leq \mu_1 \)
For the model construction we will again need some properties of the subtype relation:

27 lemma \( \Gamma \vdash \sigma \rightarrow \tau \leq \sigma' \rightarrow \tau' \implies \Gamma \vdash \sigma' \leq \sigma \) and \( \tau \leq \tau' \)

\( \Gamma \vdash \Pi f \leq \Pi g \implies \Gamma, \alpha : \star \vdash f \alpha \leq g \alpha \)

\( \Gamma \vdash \mu f \leq \mu g \implies \Gamma, \alpha : \star, \beta : \star, \alpha \leq \beta \vdash f \alpha \leq g \beta \)

\( \Box \)

We prove this in the same way as we proved lemma 14. We define a relation \( \leq' \) on types. For \( \leq' \) we have the same derivation rules as for \( \leq \), except instead of \( (TRANS) \) we have the rule \( (\leq \text{TEQ}) \).
Clearly \( \Gamma \vdash \sigma \leq' \tau \implies \Gamma \vdash \sigma \leq \tau \), and by the next lemma we also have \( \Gamma \vdash \sigma \leq \tau \implies \Gamma \vdash \sigma \leq' \tau \).

28 lemma \( \leq' \) is transitive, i.e. \( \Gamma \vdash \rho \leq' \sigma \) \( \& \Gamma \vdash \sigma \leq' \tau \implies \Gamma \vdash \rho \leq' \tau \)
proof
The proof is almost the same as for lemma 15. The only difference is that we now also have the possibility that

(d) \( \rho = \nu \mu f, \sigma = \nu \mu g \) and \( \tau = \nu \mu h \).

For this case \( \Gamma \vdash \rho \leq' \tau \) is proven as for (b) and (c):
The derivations of \( \Gamma \vdash \rho \leq' \sigma \) and \( \Gamma \vdash \sigma \leq' \tau \), must both end with \( (\leq \mu) \), possibly followed by \( (\leq \text{TEQ}) \).
So \( \Gamma, \alpha : \star, \beta : \star, \gamma : \star, \beta \leq \gamma \vdash f \alpha \leq' g \beta \) and \( \Gamma, \beta : \star, \gamma : \star, \alpha \leq \beta \vdash g \beta \leq' h \gamma \).
By the induction hypothesis \( \Gamma, \alpha : \star, \gamma : \star, \alpha \leq \beta \vdash h \alpha \leq' h \gamma \), so \( \Gamma \vdash \mu f \leq' \mu h \) and hence \( \Gamma \vdash \rho \leq' \tau \).

\( \Box \)

So \( \Gamma \vdash \sigma \leq' \tau \iff \Gamma \vdash \sigma \leq \tau \), and for \( \leq' \) it is obvious that lemma 27 holds.
We will also need
29 lemma $\Gamma \vdash \mu f \leq \mu g \Rightarrow \Gamma \vdash f(\mu f) \leq g(\mu g)$
proof
Suppose $\Gamma \vdash \mu f \leq \mu g$.
Then $\Gamma, \alpha : \star, \beta : \star \vdash f\alpha \leq g\beta$, and in the derivation of this we can substitute $\mu f$ for $\alpha$ and $\mu g$ for $\beta$, which gives us $\Gamma \vdash f(\mu f) \leq g(\mu g)$
\end{proof}

We again use a term model as the submodel for the constructor expressions. We define the relation $\leq^*$ on $\text{Kind}_*$ as we did for the model construction for $A_*^0$.

30 definition ($\leq^*$)
If $a, b \in \text{Kind}_*$, then $a$ and $b$ are closed type expressions, i.e. $\langle \nu a : \star \rangle$ and $\langle \nu b : \star \rangle$, so we can define $\leq^*$ by
\begin{align*}
a \leq^* b & \iff \langle \nu a \rangle \leq \langle \nu b \rangle
\end{align*}
\end{proof}

31 lemma $\Gamma \vdash \sigma \leq \tau \iff \forall \eta \left[ \Gamma \vdash \sigma^* \models \eta^* \iff \Gamma \vdash \tau^* \models \eta \right]$
proof By induction on $\sigma$ or $\tau$.

We define a functor $\mathcal{F}$ on $\mathcal{K}$, $\mathcal{K} = [\text{Kind}_*, \text{CPO}_*, \text{Pr}]$.

32 definition ($\mathcal{F} : \mathcal{K} \rightarrow \mathcal{K}$)
The object part of $\mathcal{F}$ is defined as follows. Let $F \in \text{Obj}(\mathcal{K})$. Then $\mathcal{F}F \in \text{Obj}(\mathcal{K})$, i.e. $\mathcal{F}F$ is a functor from $\text{Kind}_*$ to $\text{CPO}_*$. The object part of $\mathcal{F}F$, is defined by
\begin{align*}
(\mathcal{F}F)a & = \text{domain}_a \\
(\mathcal{F}F)\sigma \rightarrow \tau & = \text{FS}(F\sigma, F\tau) \\
(\mathcal{F}F)\Pi f & = \text{GP}(< F(FA) \mid a \in \text{Kind}_*>) \\
(\mathcal{F}F)\mu f & = F(f(\mu f))
\end{align*}
and the morphism part of $\mathcal{F}F$, is defined by
\begin{align*}
(\mathcal{F}F)a \leq b & = \text{coerce}_{ab} \\
(\mathcal{F}F)\sigma \leftarrow \tau \leq \sigma' \rightarrow \tau' & = \text{FS}(F\sigma' \leq \sigma, F\tau \leq \tau') \\
(\mathcal{F}F)\Pi f \leq \Pi g & = \text{GP}(< F(FA) \leq gA \mid a \in \text{Kind}_*>) \\
(\mathcal{F}F)\mu f \leq \mu g & = F(f(\mu f)) \leq g(\mu g)
\end{align*}
The morphism part of $\mathcal{F}$ is defined as follows:
if $(\eta, \theta) \in \text{Hom}_\mathcal{K}(F, G)$, so $\eta : F \rightarrow G$ and $\theta : G \rightarrow F$ then $\mathcal{F}(\eta, \theta) = (\eta', \theta')$, where
\begin{align*}
(\eta'_{a}, \theta'_{a}) & = (\text{id}_{\text{domain}_a}, \text{id}_{\text{domain}_a}) \\
(\eta'_{\sigma \leftarrow \tau}, \theta'_{\sigma \leftarrow \tau}) & = \text{FS}_{\mathcal{K}}((\eta_{\sigma}, \theta_{\sigma}), (\eta_{\tau}, \theta_{\tau})) \\
(\eta'_{\Pi f}, \theta'_{\Pi f}) & = \text{GP}_{\mathcal{K}}(< (\eta_{Fa}, \theta_{Fa}) \mid a \in \text{Kind}_*>) \\
(\eta'_{\mu f}, \theta'_{\mu f}) & = (\eta(f(\mu f)), \theta(f(\mu f)))
\end{align*}
That $\mathcal{F}$ preserves identities and composition can easily be verified (pointwise).
\end{proof}

In the same way lemma 24 is proved, we can prove that $\mathcal{F}$ is $\omega$-continuous.
So $\mathcal{F}$ has an initial fixed point $(\text{Dom} \& \text{Coe}, (\Phi, \Psi))$, which gives us a family of epos solving the recursive domain equations with the associated bijections, and a family of coercions satisfying the coherence conditions.
For the coercions between recursive types

\[ \text{Coe}_{\mu f}(\mu g) = \Phi^{-1}_{g} \circ \text{Coe}_{f(\mu f)}(\mu g) \circ \Phi_{\mu f} \quad (\star) \]

will hold. This means that coercions commute with unfolding and folding, i.e. \( \text{Coe}_{\mu f}(\mu g) \) followed by \( \text{unfold}_{\mu g} \) gives the same result as \( \text{unfold}_{\mu f} \) followed by \( \text{Coe}_{f(\mu f)}(\mu g) \); and \( \text{fold}_{\mu f} \) followed by \( \text{Coe}_{\mu f}(\mu g) \) gives the same result as \( \text{Coe}_{f(\mu f)}(\mu g) \) followed by \( \text{fold}_{\mu g} \). However, because of the subscripts of \( \text{fold} \) and \( \text{unfold} \) this is not needed for coherence; \( \text{Coe}_{\mu f}(\mu g) \) and \( \text{Coe}_{f(\mu f)}(\mu g) \) could be completely unrelated.

Because of (\( \star \)), the subscript of \( \text{unfold} \) can be omitted. We can check that lemma 38 holds for the rule (\( \text{UNFOLD} \)), so the semantics will then still be coherent.

Possibly the subscript of \( \text{fold} \) can also be omitted. However, as shown in the example on page 16, there are two possible types for a term \( \text{fold} M \), and \( \text{fold} M \) does not have a minimal type. Therefore the coherence proof as given in the appendix can not be used.
5.4 The construction of a cpo model for $\Lambda \leq \mu_3$

We distinguish covariant and contravariant positions in trees. A node or leaf in $t$ is at a covariant position in $t$ if, going from that node or leaf to the root of $t$, we enter an even number of $-$ nodes from the left-hand side, and else it is at a contravariant position in $t$. For example, in

\[
\begin{align*}
\text{int} & \longrightarrow \alpha \\
\text{real} & \text{int} \longrightarrow \text{real}
\end{align*}
\]

int and $\alpha$ occur at covariant positions, whereas real and

\[
\begin{align*}
\text{int} & \longrightarrow \text{real} \\
\text{int} & \longrightarrow \text{real}
\end{align*}
\]

occur at contravariant positions.

33 definition ($\leq^*$)

$s \leq^* t$

iff

except for their leaves, $s$ and $t$ are the same tree, and for all leaves $a$ and $b$ in the same place in $s$ and $t$, respectively:

- $a \leq^* b$ and $a$ and $b$ occur at covariant positions in $s$ and $t$, or
- $b \leq^* a$ and $a$ and $b$ occur at contravariant positions in $s$ and $t$, or
- $a \equiv b$

\(\Box\)

We want to prove

$$\Gamma \vdash \sigma \leq \tau \iff \forall \eta [\Gamma \vdash \sigma : \star ] \eta \leq^* [\Gamma \vdash \tau : \star ] \eta$$

It is really the implication $\Rightarrow$ that is important, since if that implication holds, then a family of coercion functions

$$< \text{Coe}_{a,b} | a \leq^* b >$$

will contain the required coercions.

34 lemma $\Gamma \vdash \sigma \leq \tau \Rightarrow \forall \eta [\Gamma \vdash \sigma : \star ] \eta \leq^* [\Gamma \vdash \tau : \star ] \eta$

proof by induction on the derivation of $\Gamma \vdash \sigma \leq \tau$.

We only treat the prime case, $(\leq \mu)$. Suppose the last rule of the derivation is $(\leq \mu)$,

$$\begin{align*}
\Gamma, \alpha : *, \beta : *, \alpha \leq \beta \vdash f \alpha \leq g \beta \\
\Gamma \vdash \mu f \leq \mu g
\end{align*}$$

Define $\Gamma' = \Gamma, \alpha : *, \beta : *, \alpha \leq \beta$.

By the induction hypothesis:

$$\forall \eta [\Gamma' \vdash f \alpha : \star ] \eta \leq^* [\Gamma' \vdash g \beta : \star ] \eta.$$

To prove:

$$\forall \eta [\Gamma \vdash \mu f : \star ] \eta \leq^* [\Gamma \vdash \mu g : \star ] \eta.$$

Assume $\Gamma \vdash \eta$. Define $F = \Gamma \vdash f : \star \Rightarrow \star ] \eta$ and $G = \Gamma \vdash g : \star \Rightarrow \star ] \eta$.

By induction on $i \in \mathbb{N}$ we prove $F^i \perp \leq^* G^i \perp$.

base $F^0 \perp = \perp \leq^* \perp = G^0 \perp$
step \( F^{i+1} \perp = (\Gamma \vdash f : * \Rightarrow * \eta)(F^{i} \perp) = [\Gamma' \vdash f' \alpha : * \eta[\alpha := F^{i} \perp][\beta := G^{i} \perp] \]
\( G^{i+1} \perp = (\Gamma \vdash g : * \Rightarrow * \eta)(G^{i} \perp) = [\Gamma' \vdash g' \beta : * \eta[\alpha := F^{i} \perp][\beta := G^{i} \perp] \]
\( F^{i} \perp \leq^{*} G^{i} \perp, \) so \( \eta[\alpha := F^{i} \perp][\beta := G^{i} \perp] \) satisfies \( \Gamma' \). Then by the induction hypothesis
\( [\Gamma' \vdash f' \alpha : * \eta[\alpha := F^{i} \perp][\beta := G^{i} \perp] \leq^{*} [\Gamma' \vdash g' \beta : * \eta[\alpha := F^{i} \perp][\beta := G^{i} \perp] \)
so \( F^{i+1} \perp \leq^{*} G^{i+1} \perp. \)

\( [\Gamma \vdash \mu f : * \eta = \bigsqcup F^{i} \perp \leq^{*} \bigsqcup G^{i} \perp = [\Gamma \vdash \mu g : * \eta]. \)

\( \square \)

It is easy to see that

\[
\begin{array}{ccc}
  s & \leq^{*} & t \\
\Pi_\alpha & \leq^{*} & \Pi_\alpha \\
\downarrow & & \downarrow \\
  s & & t
\end{array}
\]

so

\[
\begin{align*}
  [\sigma \rightarrow \tau] \leq^{*} & [\sigma' \rightarrow \tau'] & \Rightarrow & [\sigma'] \leq^{*} [\sigma ] \text{ and } [\tau'] \leq^{*} [\tau ] \\
  [\Pi f ] & \leq^{*} [\Pi g ] & \Rightarrow & \forall \alpha \in \text{Kind}, [f\alpha : * ] \leq^{*} [g\alpha : * ]
\end{align*}
\]

By \( P_2 \) : for all \([\sigma \rightarrow \tau] \leq^{*} [\sigma' \rightarrow \tau'] \)
\( \text{Coeq} [\sigma \rightarrow \tau][\sigma' \rightarrow \tau'] = \Phi^{-1}_{[\sigma \rightarrow \tau][\sigma' \rightarrow \tau']} \Phi_{[\sigma \rightarrow \tau] \circ \text{FS}(\text{Coeq} \{ \sigma \} \circ \{ \sigma \}) \circ \text{FS}([\sigma])}
\)
and by \( P_3 \) : for all \([\Pi f ] \leq^{*} [\Pi g ] \)
\( \text{Coeq} [\Pi f][\Pi g ] = \Phi^{-1}_{[\Pi g]} \Phi_{[\Pi f ] \circ \text{GP}(< \text{Coeq} \{ f \} \circ \{ \beta \} \mid \{ \alpha \} \in \text{Kind} >) \circ \text{FS}([\Pi f ]}
\)

To construct the required family of cpos and a family of coercion functions we can now use the same construction we used for \( A \leq^{*} \).

35 definition \( (F : \mathcal{K} \rightarrow \mathcal{K}) \)

The object part of \( F \) is defined as follows. Let \( F \in \text{Obj}(\mathcal{K}) \). Then the object part of \( FF \), a mapping from \( \text{Obj}(\text{Kind}_* \) to \( \text{Obj}(\text{CPO}_\perp \) is defined by

\[
\begin{array}{ccc}
  (FF) \sigma & = & \text{domain}_\sigma \\
  (FF) & \sigma \rightarrow \tau & = \text{FS}(\text{Fr}, \text{Fr}) \\
  (FF) \Pi_\alpha & = & \text{GP}(< F[\tau[\alpha := a]] \mid a \in \text{Kind}_* >) \\
  (FF) \perp & = & F \perp
\end{array}
\]

and the morphism part of \( FF \), a mapping from \( \text{Mor}(\text{Kind}_* \) to \( \text{Mor}(\text{CPO}_\perp \) is defined by

\[
\begin{array}{ccc}
  (FF) \sigma \leq \tau & = & \text{coerce}_{\sigma \rightarrow \tau} \\
  (FF) & \sigma \rightarrow \tau & = \text{FS}(\sigma' \leq \sigma, \tau' \leq \tau) \\
  (FF) \Pi_\alpha \leq \Pi_\alpha & = & \text{GP}(< F[\tau[\alpha := a]] \leq [\tau[\alpha := a]] \mid a \in \text{Kind}_* >) \\
  (FF) \perp \leq \perp & = & F \perp \leq \perp
\end{array}
\]

39
The morphism part of $\mathcal{F}$ is defined as follows: if $(\eta, \theta) \in \text{Hom}_{\mathcal{K}}(F, G)$, then $\mathcal{F}(\eta, \theta) = (\eta', \theta')$, where

$$(\eta'_\sigma, \theta'_\sigma) = (id\text{\_domain}_\sigma, id\text{\_domain}_\sigma)$$

$$(\eta'_\sigma, \theta'_\sigma) = FSPR((\eta_\sigma, \theta_\sigma), (\eta_\tau, \theta_\tau))$$

$$(\eta_{\frac{1}{\sigma}}, \theta_{\frac{1}{\sigma}}) = GP_R(< \eta_{\sigma[a = \bot]}, \theta_{\sigma[a = \bot]} | a \in \text{Kind}_*>)$$

$$(\eta_{\frac{1}{\bot}}, \theta_{\frac{1}{\bot}}) = (\eta_{\bot}, \theta_{\bot})$$

$\Box$

In the same way lemma 24 is proved, we can prove that $\mathcal{F}$ is $\omega$-continuous. So $\mathcal{F}$ has an initial fixed point $(\text{Dom}\&\text{Coe}, (\mathcal{I}, \mathcal{I}))$ which gives us a family of cpos solving the recursive domain equations with the associated bijections, and a family of coercions satisfying the coherence conditions.

5.5 The construction of a cpo model for $\Lambda \leq \mu_2$

The only difficulty for $\Lambda \leq \mu_2$ is that to construct a model we have to prove

$$\Gamma \vdash \sigma \rightarrow \tau \leq \sigma' \rightarrow \tau' \implies \Gamma \vdash \sigma' \leq \sigma \text{ and } \tau \leq \tau'$$

$$\Gamma \vdash \Pi f \leq \Pi g \implies \Gamma, \alpha : \ast \vdash f\alpha \leq g\alpha$$

which is still an open problem.

Once it is proved, we can construct a model in the same way the models for $\Lambda \leq \mu_1$ and $\Lambda \leq \mu_3$ have been constructed.

The subtype relation on $\text{Kind}_*$ is of course defined as it was for $\Lambda \leq \mu_1$ (definition 30).
6 Conclusion

The theory of O-categories has proved extremely useful. Because the functor category \([A, B]\) is an O-category if \(B\) is, we can use all the standard results for O-categories and the associated categories of embedding-projection pairs. The fact that we have used the O-category \(CPO\) is not essential. Other O-categories could be used, for instance the category of directed cpos or complete lattices: types would then be interpreted as directed cpos or complete lattices.

To all the systems we described, other type constructors, such as \(\times\) (Cartesian product), \(+\) (separated sum), \(\circ\) (smashed product), \(\oplus\) (coalesced sum) or \((\quad)\) (lifting) can easily be included. We add them as constructor constants of the appropriate kind, and add the associated type inference rules. For the general model definitions the necessary domain equations must be given, and all that is required for the construction of a cpo model is a corresponding functor, like we have the function space functor \(FS\) for \(\rightarrow\)-types.

For example, for \(\times\)-types we would have to add a type constructor \(\times\) of kind \(* \Rightarrow * \Rightarrow *\) and the recursive domain equations
\[
\text{Dom}_{[\sigma \times \tau]} \cong \text{Dom}_{[\sigma]} \times \text{Dom}_{[\tau]}
\]
so we would have to extend the definition of \(F\) with
\[
F_{[\sigma \times \tau]}(<D_{\alpha}| a \in \text{Kind }>) = CP(D_{[\sigma]}, D_{[\tau]})
\]
where \(CP\) is the product functor. The natural subtyping rule for \(\times\)-types
\[
\frac{\Gamma \vdash \sigma \leq \sigma' \quad \Gamma \vdash \tau \leq \tau'}{\Gamma \vdash \sigma \times \tau \leq \sigma' \times \tau'}
\]
can be added, and for coherence we will need the additional requirement
\[
\text{CoF}_{[\sigma \times \tau]} \equiv \Phi_{[\sigma \times \tau]}^{-1} \circ CP(\text{CoF}_{[\sigma]} \times \text{CoF}_{[\tau]} \circ \Phi_{[\sigma \times \tau]})
\]
The type constructor \(\Sigma\), which can be used for abstract data types (see [MP88]), can also be added. These \(\Sigma\)-types or existential types, can be treated like the \(\Pi\)-types. Just like the generalized product functor is used for \(\Pi\)-types, the generalized sum functor (see [tEH98]) can be used for \(\Sigma\)-types.

For the systems with subtyping, interesting extensions are of course labelled products, i.e. records, and bounded quantification.

For bounded quantification we have the type formation rule
\[
\frac{\Gamma, \alpha : *, \alpha \leq \sigma \vdash \tau : * \quad \Gamma \vdash \sigma : *}{\Gamma \vdash (\Pi \alpha \leq \sigma. \tau) : *}
\]
The recursive domain equations for such a type is
\[
\text{Dom}_{[\Gamma, \alpha : *, \alpha \leq \sigma. \tau]} \equiv \prod_{\alpha \in \text{Kind }\cdot} \text{Dom}_{[\Gamma, \alpha : *, \alpha \leq \sigma]}\]
so we get
\[
\text{Dom}_{[(\Pi \alpha \leq \sigma. \tau)]} \cong GP(<\text{Dom}_{[\tau]}| \{\alpha\} \in \text{Kind }\cdot \quad \{\alpha\} \leq^* \{\sigma\}>)
\]
The subtyping rule for II-types becomes

\[
\Gamma, \alpha : \tau, \alpha \leq \sigma \vdash \tau \leq \tau' \quad \Gamma \vdash \sigma' \leq \sigma \quad (\leq II) \\
\Gamma \vdash (\Pi \alpha : \sigma. \tau) \leq (\Pi \alpha : \sigma'. \tau')
\]

and for the coercion functions we get the following coherence conditions

\[
\text{Coef}_{\{(\Pi \alpha : \sigma. \tau)\} \mid (\Pi \alpha : \sigma'. \tau')\}} = \Phi_{\{(\Pi \alpha : \sigma. \tau)\} \mid (\Pi \alpha : \sigma'. \tau')\}} \circ GP(\text{Coef}_{\{\alpha \} \mid \{\alpha'\} \leq \{\tau'\} >}) \circ \text{proj}_{a} \mid a \in \text{Kind}_{\alpha}, \alpha \leq \{\sigma'\} > \circ \Phi_{\{(\Pi \alpha : \sigma. \tau)\}}
\]

where \(\text{proj}_{a}\) is the \(\mathit{a}\)-th projection function, so

\[
\text{proj}_{a} \in (\prod_{\sigma \leq \{\tau\}} \text{Dom}_{\tau} \mid \{a = a\}) \rightarrow \text{Dom}_{\tau} \mid \{a = a\}
\]

and

\[
\langle \text{proj}_{a} \mid a \leq \{\sigma\} > \in (\prod_{\sigma \leq \{\tau\}} \text{Dom}_{\tau} \mid \{a = a\}) \rightarrow (\prod_{\sigma' \leq \{\tau'\}} \text{Dom}_{\tau} \mid \{a = a\})
\]

Labelled products can be handled similarly. For these types we have the type formation rule

\[
\Gamma \vdash \sigma_1 : \tau_1, \ldots, \sigma_n : \tau_n \mid l_1, \ldots, l_n \in \mathcal{L} \quad \forall i,j (l_i = l_j \Rightarrow i = j) \\
\Gamma \vdash \langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle >
\]

Here \(\mathcal{L}\) is the set of all labels.

The required domain equations are

\[
\text{Dom}_{\{\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle \}} \equiv \prod_{l_i \in \{l_1, \ldots, l_n\}} \text{Dom}_{\{\sigma_i : \tau_i\}}
\]

so we get

\[
\text{Dom}_{\{\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle \}} \equiv \text{GP}(\langle \text{Dom}_{\sigma} \mid l_i \in \{l_1, \ldots, l_n\} \rangle)
\]

The subtyping rule for record-types is

\[
\Gamma \vdash \sigma_1 \leq \tau_1, \ldots, \sigma_m \leq \tau_m \quad m \leq n \\
\Gamma \vdash \langle l_1 : \sigma_1, \ldots, l_m : \sigma_m \rangle \leq \langle l_1 : \tau_1, \ldots, l_m : \tau_m \rangle \quad (\leq \text{REC})
\]

and the associated coherence conditions are

\[
\Phi_{\langle l_1 : \sigma_1, \ldots, l_m : \sigma_m \rangle} \circ \text{Coef}_{\{\langle l_1 : \sigma_1, \ldots, l_m : \sigma_m \rangle \}} = \Phi_{\langle l_1 : \sigma_1, \ldots, l_m : \sigma_m \rangle} \circ \text{GP}(\langle \text{Coef}_{\{\sigma_i \mid \{l_i = l_i\} \rangle} \rangle \circ \text{proj}_{i} \mid l_i \in \{l_1, \ldots, l_m\} > \circ \Phi_{\langle l_1 : \sigma_1, \ldots, l_m : \sigma_m \rangle}
\]

where

\[
\langle \text{proj}_{i} \mid l_i \in \{l_1, \ldots, l_m\} > \in (\prod_{l_i \in \{l_1, \ldots, l_m\}} \text{Dom}_{\sigma_i} \mid \{1\}) \rightarrow (\prod_{l_i \in \{l_1, \ldots, l_m\}} \text{Dom}_{\sigma_i} \mid \{1\})
\]

When record types are added in this way, the models will also provide the semantics for record updates. It remains to be seen, which of the operations on records and record types mentioned in [CM89] can be modelled in this way.
Labelled sums, or variants, and bounded \(\Sigma\)-types can be treated in the same way as bounded \(\Pi\)-types and labelled products. Instead of the generalized product functor \(GP\) we use the generalized sum functor.

Another possible extension of the systems is to allow abstraction not only of terms over types but of terms over all kinds, and the corresponding form of application, i.e. terms to kinds. The system we then get is \(F\omega\) (\(\lambda\omega\) in Barendregt's cube [Bar9]) extended with subtyping and recursive types (but without recursion on higher kinds). To model the polymorphic types \(\Pi(\lambda\alpha : \kappa.\sigma)\) we also use the \(GP\)-functor, only this time applied to a family indexed by \(\text{Kind}_\kappa\) instead of \(\text{Kind}_\omega\).

**Acknowledgements**

I would like to thank Marc Bezem, Huub ten Eikelder and Kees Hemerik for helpful advice and discussions on this paper.
Appendix : Coherence

We will now prove that the semantics is coherent if the coherence conditions $P_0, P_1, P_2$ and $P_3$ hold. We use the fact that we have minimal typing in $A_\Sigma$:

36 lemma (minimal typing)
In a given context $\Gamma$ every term $M$ has a minimal type, i.e. a type $\sigma_{\min}$ such that

$$\Gamma \vdash M : \sigma_{\min} \quad \text{and} \quad \forall \sigma \quad \Gamma \vdash M : \sigma \Rightarrow \Gamma \vdash \sigma_{\min} \leq \sigma$$

proof by induction on $M$

Type derivations for a term are for a large part determined by the syntax of that term. If we have a derivation for $\Gamma \vdash M : \sigma$, then the syntax of $M$ determines which is the last rule other than $(\text{SUB})$ used in that derivation. For instance, if $\Gamma \vdash (Ax : \sigma.M) : \sigma \rightarrow \tau$ the last rule other than $(\text{SUB})$ used in the derivation must be $(\rightarrow I)$. We cannot tell by the syntax of a term if and where the rules $(\text{SUB})$ may have been used in a type derivation.

First a few words about notation.

- By $\llbracket \Gamma \vdash M : \sigma \rrbracket$ we mean the function $(\lambda \eta . \llbracket \Gamma \vdash M : \sigma \rrbracket \eta)$ from environments $\eta, \eta \models \Gamma$, to $\bigcup_{\eta} \text{Dom}_{\Gamma}^{\tau} \eta$.

- Suppose $\Delta$ is a derivation deriving $\Gamma \vdash M : \tau$ from $\Gamma_1 \vdash N_1 : \sigma_1 \ldots \Gamma_n \vdash N_n : \sigma_n$, i.e.

$$\begin{align*}
\Gamma_1 &\vdash N_1 : \sigma_1 & \ldots & \Gamma_n &\vdash N_n : \tau_n \\
& \vdots & & \vdots \\
\Gamma &\vdash M : \tau
\end{align*}$$

Using the definition of $\llbracket \rrbracket$, this derivation gives us $\llbracket \Gamma \vdash M : \tau \rrbracket$ in terms of $\llbracket \Gamma \vdash N_1 : \sigma_1 \rrbracket \ldots \llbracket \Gamma \vdash N_n : \sigma_n \rrbracket$. In other words, $\Delta$ determines a function $R_\Delta$ such that

$$\llbracket \Gamma \vdash M : \tau \rrbracket = R_\Delta(\llbracket \Gamma \vdash N_1 : \sigma_1 \rrbracket \ldots \llbracket \Gamma \vdash N_n : \sigma_n \rrbracket)$$

- We write

$$\begin{array}{c}
\Gamma \vdash M : \sigma \\
\hline
\Gamma \vdash M : \tau
\end{array}$$

for any derivation deriving $\Gamma \vdash M : \sigma$ from $\Gamma \vdash M : \tau$. Such a derivation can only use rule $(\text{SUB})$, a number of times.

- If $(T)$ is a type inference rule, we write

$$\begin{array}{c}
\Gamma_1 \vdash N_1 : \sigma_1 \ldots \Gamma_n \vdash N_n : \sigma_n \\
\hline
\Gamma \vdash M : \tau
\end{array} (T)$$

If $\Gamma \vdash M : \tau$ can be derived from $\Gamma_1 \vdash N_1 : \sigma_1 \ldots \Gamma_n \vdash N_n : \sigma_n$ using $(T)$ exactly once, $(\text{SUB})$ any number of times, and no other rules, i.e.

$$\begin{array}{c}
\Gamma_1 \vdash N_1 : \sigma_1 \ldots \Gamma_n \vdash N_n : \sigma_n \\
\hline
\Gamma_1 \vdash N_1 : ? \ldots \Gamma_n \vdash N_n : ? (T) \\
\hline
\Gamma \vdash M : \tau
\end{array}$$

44
37 lemma For all derivations $\Delta$:

$$\Gamma \vdash M : \sigma \quad \Gamma \vdash M : \tau$$

$R_\Delta$ is the same, viz. $R_\Delta = \lambda \xi. \text{Coe}_\sigma \circ \xi$

proof follows directly from $P_0$ and $P_1$.

38 lemma For all type inference rules $(T)$ not equal to $(\text{SUB})$ all derivations $\Delta$,

$$\Delta : \frac{\Gamma_1 \vdash N_1 : \sigma_1 \ldots \Gamma_n \vdash N_n : \sigma_n}{\Gamma \vdash M : \tau}$$

yield the same $R_\Delta$.

proof We distinguish between the four possible choices for $(T)$: $(\rightarrow I)$, $(\rightarrow E)$, $(III)$ and $(II)$. For the first two we will need $P_2$, for the last two $P_3$. We treat only one case, $\rightarrow E$; the others are similar.

Suppose

$$\Delta : \frac{\Gamma \vdash M : \sigma_1 \rightarrow \sigma_2 \quad \Gamma \vdash N : \sigma_3}{\Gamma \vdash MN : \tau}$$

then there are types $\rho_1$ and $\rho_2$ such that $\sigma_3 \leq \rho_1 \leq \sigma_1$ and $\sigma_2 \leq \rho_2 \leq \tau$ and

$$\Delta : \frac{M : \sigma_1 \rightarrow \sigma_2 \quad N : \sigma_3}{\frac{\rho_1 \rightarrow p_2 \quad N : p_1}{\rho_1}(\rightarrow E)}$$

Using $P_2$, we can prove that $R_\Delta$ does not depend on $\rho_1$ and $\rho_2$.

$$\Phi_{\rho_1 \rightarrow \rho_2} \left[ M : \rho_1 \rightarrow \rho_2 \right] \eta$$

= \{ lemma 37\}

$$\Phi_{\rho_1 \rightarrow \rho_2} \left( \text{Coe}_{\sigma_1 \rightarrow \sigma_2} \rho_1 \rightarrow \rho_2 \left[ M : \sigma_1 \rightarrow \sigma_2 \right] \eta \right)$$

= \{ $P_2$\}

$$\Phi_{\rho_1 \rightarrow \rho_2} \left( (\Phi_{\rho_1 \rightarrow \rho_2}^{-1} \circ \text{FS} \left( \text{Coe}_{\rho_1, \sigma_1} \circ \text{Coe}_{\rho_2, \sigma_2} \circ \Phi_{\rho_1 \rightarrow \rho_2} \right)) \left[ M : \sigma_1 \rightarrow \sigma_2 \right] \eta \right)$$

= \{ definition $FS$\}

$$\Phi_{\rho_1 \rightarrow \rho_2} \left( (\Phi_{\rho_1 \rightarrow \rho_2}^{-1} \circ \text{Coe}_{\sigma_2 \rightarrow \sigma_2} \circ (\Phi_{\sigma_1 \rightarrow \sigma_2} \circ \text{Coe}_{\rho_1, \sigma_1} \left[ M : \sigma_1 \rightarrow \sigma_2 \right] \eta)) \circ \text{Coe}_{\rho_1, \alpha} \right)$$

= \{ $\Phi_{\rho_1 \rightarrow \rho_2}$ is a bijection\}

$$\text{Coe}_{\sigma_2 \rightarrow \sigma_2} \circ (\Phi_{\sigma_1 \rightarrow \sigma_2} \circ \text{Coe}_{\rho_1, \alpha} \left[ M : \sigma_1 \rightarrow \sigma_2 \right] \eta) \circ \text{Coe}_{\rho_1, \alpha}$$

and using this we can prove
\[ MN : \tau \] \\
= \{ \text{lemma 37} \} \\
Coe_{\rho_2} \tau \left[ MN : \rho_2 \right] \eta \\
= \{ \text{definition [ ] for } (\rightarrow E) \} \\
Coe_{\rho_2} \tau ( \Phi_{\rho_1 \rightarrow \rho_2} \left[ M : \rho_1 \rightarrow \rho_2 \right] \eta ) \left[ N : \rho_1 \right] \eta \\
= \{ \text{lemma 37} \} \\
Coe_{\rho_2} \tau ( \Phi_{\rho_1 \rightarrow \rho_2} \left[ M : \rho_1 \rightarrow \rho_2 \right] \eta ) ( \text{Coe}_{\sigma_3} \left[ N : \sigma_3 \right] \eta ) \\
= \\
\{ \text{see above} \} \\
( \text{Coe}_{\rho_2} \tau \circ \text{Coe}_{\sigma_2} \rho_2 \circ (\Phi_{\sigma_1 \rightarrow \sigma_2} \left[ M : \sigma_1 \rightarrow \sigma_2 \right] \eta ) \circ \text{Coe}_{\sigma_1} \rho_1 \circ \text{Coe}_{\sigma_3} \rho_1 ) \left[ N : \sigma_3 \right] \eta \\
= \{ 2 \times P_1 \} \\
( \text{Coe}_{\sigma_2} \tau \circ \text{Coe}_{\sigma_2} \rho_2 \circ (\Phi_{\sigma_1 \rightarrow \sigma_2} \left[ M : \sigma_1 \rightarrow \sigma_2 \right] \eta ) \circ \text{Coe}_{\sigma_1} \rho_1 ) \left[ N : \sigma_3 \right] \eta \\
\text{So } [ MN : \tau ] = \Delta \eta [ MN : \tau ] \eta \text{ does not depend on } \rho_1 \text{ or } \rho_2. \\
\Box

39 \text{ theorem (coherence)}

All derivations of } \Gamma \vdash M : \tau \text{ give the same meaning } [ \Gamma \vdash M : \tau ] \eta .

\text{proof by induction on } M .

\text{base} \\
M \text{ is a variable or a constant } : \text{ trivial}.

\text{step} \\
Suppose we have two derivations, } \Delta_1 \text{ and } \Delta_2 \text{, for } \Gamma \vdash M : \tau . \text{ Then these derivations must end with the same rule, so they are of the following form}

\[
\Delta_1 : \quad \begin{array}{c}
\Delta_{11} \\
\Delta_{1n}
\end{array}
\begin{array}{c}
\Gamma_1 \vdash N_1 : \sigma_1 \\
\vdots \\
\Gamma_n \vdash N_n : \sigma_n (T)
\end{array}
\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \tau}
\Delta_2 : \quad \begin{array}{c}
\Delta_{21} \\
\Delta_{2n}
\end{array}
\begin{array}{c}
\Gamma_1 \vdash N_1 : \rho_1 \\
\vdots \\
\Gamma_n \vdash N_n : \rho_n (T)
\end{array}
\frac{\Gamma \vdash M : \rho}{\Gamma \vdash M : \tau}
\]

By the induction hypothesis, all derivations for } \Gamma \vdash N_i : \sigma_i \text{ yield the same meaning } [ \Gamma \vdash N_i : \sigma_i ] \text{, and the same is true for } \Gamma \vdash N_i : \rho_i .

So in } \Delta_1 \text{ each } \Delta_{ij} \text{ can be replaced by any derivation we want, and the resulting derivation will give the same meaning for } \Gamma \vdash M : \tau \text{ as } \Delta_j .

\text{We will now use the fact that we have minimal typing.}

Let } \alpha_i \text{ be the minimal type of } N_i \text{ for } i = 1 \ldots n . \text{ Then the following two derivations, } \Delta_1' \text{ and } \Delta_2' , \text{ give the same meaning for } \Gamma \vdash M : \tau \text{ as } \Delta_1 \text{ and } \Delta_2 , \text{ respectively:}

\[
\Delta_1' : \quad \begin{array}{c}
\Delta_{11}' \\
\Delta_{1n}'
\end{array}
\begin{array}{c}
\Gamma_1 \vdash N_1 : \alpha_1 \\
\vdots \\
\Gamma_n \vdash N_n : \alpha_n (T)
\end{array}
\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \tau}
\Delta_2' : \quad \begin{array}{c}
\Delta_{21}' \\
\Delta_{2n}'
\end{array}
\begin{array}{c}
\Gamma_1 \vdash N_1 : \rho_1 \\
\vdots \\
\Gamma_n \vdash N_n : \rho_n (T)
\end{array}
\frac{\Gamma \vdash M : \rho}{\Gamma \vdash M : \tau}
\]

But by lemma 38 for all derivations } \Delta 

\[
\Delta : \quad \begin{array}{c}
\Gamma_1 \vdash N_1 : \alpha_1 \ldots \Gamma_n \vdash N_n : \alpha_n (T)
\end{array}
\frac{\Gamma \vdash M : \tau}{\Gamma \vdash M : \tau}
\]

\(R_\Delta\) is the same. So } \Delta_1' \text{ and } \Delta_2' \text{ both give the same meaning for } \Gamma \vdash M : \tau .

\(\Box\)
Using lemma 16 and the examples on page 27, we can actually show that the semantics is coherent if and only if $P_0$, $P_1$, $P_2$, and $P_3$ hold.
References


49
In this series appeared:

<table>
<thead>
<tr>
<th>Volume</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>89/1</td>
<td>E.Zs.Lepoeter-Molnar</td>
<td>Reconstruction of a 3-D surface from its normal vectors.</td>
</tr>
<tr>
<td>89/2</td>
<td>R.H. Mak, P.Struik</td>
<td>A systolic design for dynamic programming.</td>
</tr>
<tr>
<td>89/3</td>
<td>H.M.M. Ten Eikelder, C. Hemerik</td>
<td>Some category theoretical properties related to a model for a polymorphic lambda-calculus.</td>
</tr>
<tr>
<td>89/4</td>
<td>J.Zwiers, W.P. de Roever</td>
<td>Compositionality and modularity in process specification and design: A trace-state based approach.</td>
</tr>
<tr>
<td>89/5</td>
<td>Wei Chen, T.Verhoeff, J.T.Udding</td>
<td>Networks of Communicating Processes and their (De-)Composition.</td>
</tr>
<tr>
<td>89/6</td>
<td>T.Verhoeff</td>
<td>Characterizations of Delay-Insensitive Communication Protocols.</td>
</tr>
<tr>
<td>89/7</td>
<td>P.Struik</td>
<td>A systematic design of a parallel program for Dirichlet convolution.</td>
</tr>
<tr>
<td>89/9</td>
<td>K.M. van Hee, P.M.P. Rambags</td>
<td>Discrete event systems: Dynamic versus static topology.</td>
</tr>
<tr>
<td>89/10</td>
<td>S.Ramesh</td>
<td>A new efficient implementation of CSP with output guards.</td>
</tr>
<tr>
<td>89/11</td>
<td>S.Ramesh</td>
<td>Algebraic specification and implementation of infinite processes.</td>
</tr>
<tr>
<td>89/12</td>
<td>A.T.M.Aerts, K.M. van Hee</td>
<td>A concise formal framework for data modeling.</td>
</tr>
<tr>
<td>89/14</td>
<td>H.C.Haesen</td>
<td>ELDA, data manipulatie taal.</td>
</tr>
<tr>
<td>89/15</td>
<td>J.S.C.P. van der Woude</td>
<td>Optimal segmentations.</td>
</tr>
<tr>
<td>89/16</td>
<td>A.T.M.Aerts, K.M. van Hee</td>
<td>Towards a framework for comparing data models.</td>
</tr>
<tr>
<td>89/17</td>
<td>M.J. van Diepen, K.M. van Hee</td>
<td>A formal semantics for Z and the link between Z and the relational algebra.</td>
</tr>
</tbody>
</table>
Formal methods and tools for the development of distributed and real time systems, p. 17.

Dynamic process creation in high-level Petri nets, pp. 19.

Foundations of Compositional Program Refinement - safety properties -, p. 38.

Decomposition of delay-insensitive circuits, p. 25.

On the delay-sensitivity of gate networks, p. 23.


A logic for one-pass, one-attributed grammars, p. 14.

Receptive Process Theory, p. 16.

Combining the functional and the relational model, p. 15.

A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).

A proof system for process creation, p. 84.

A proof theory for a sequential version of POOL, p. 110.

Proving termination of Parallel Programs, p. 7.

A proof system for the language POOL, p. 70.

Compositionality in the temporal logic of concurrent systems, p. 17.

A fully abstract model for concurrent logic languages, p. 23.

On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.
90/18 J. Coenen  
E. v. d. Sluis  
E. v. d. Velden  

Design and implementation aspects of remote procedure calls, p. 15.

90/19 M. M. de Brouwer  
P. A. C. Verkoulen  

Two Case Studies in ExSpect, p. 24.

90/20 M. Rem  

The Nature of Delay-Insensitive Computing, p. 18.

90/21 K. M. van Hee  
P. A. C. Verkoulen  

Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.

91/01 D. Alstein  


91/02 R. P. Nederpelt  
H. C. M. de Swart  

Implication. A survey of the different logical analyses "if...then...", p. 26.

91/03 J. P. Katoen  
L. A. M. Schoenmakers  

Parallel Programs for the Recognition of $P$-invariant Segments, p. 16.

91/04 E. v. d. Sluis  
A. F. v. d. Stappen  

Performance Analysis of VLSI Programs, p. 31.

91/05 D. de Reus  

An Implementation Model for GOOD, p. 18.

91/06 K. M. van Hee  

SPECIFICATIEMETHODEN, een overzicht, p. 20.

91/07 E. Poll  

CPO-models for second order lambda calculus with recursive types and subtyping, p.

91/08 H. Schepers  

Terminology and Paradigms for Fault Tolerance, p. 25.

91/09 W. M. P. v. d. Aalst  

Interval Timed Petri Nets and their analysis, p. 53.

91/10 R. C. Backhouse  
P. J. de Bruin  
P. Hoogendijk  
G. Malcolm  
E. Voermans  
J. v. d. Woude  

POLYNOMIAL RELATORS, p. 52.

91/11 R. C. Backhouse  
P. J. de Bruin  
G. Malcolm  
E. Voermans  
J. van der Woude  

Relational Catamorphism, p. 31.

91/12 E. van der Sluis  


91/13 F. Rietman  

A note on Extensionality, p. 21.

91/14 P. Lemmens  

The PDB Hypermedia Package. Why and how it was built, p. 63.
<table>
<thead>
<tr>
<th>Page</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>91/16</td>
<td>A.J.J.M. Marcelis</td>
<td>An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.</td>
</tr>
<tr>
<td>91/18</td>
<td>Rik van Geldrop</td>
<td>Transformational Query Solving, p. 35.</td>
</tr>
<tr>
<td>91/19</td>
<td>Erik Poll</td>
<td>Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.</td>
</tr>
<tr>
<td>91/23</td>
<td>K.M. van Hee, L.J. Somers, M. Voorhoeve</td>
<td>Z and high level Petri nets, p. 16.</td>
</tr>
<tr>
<td>91/25</td>
<td>P. Zhou, J. Hooman, R. Kuiper</td>
<td>A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness, p. 52.</td>
</tr>
<tr>
<td>91/27</td>
<td>F. de Boer, C. Palamidessi</td>
<td>Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.</td>
</tr>
<tr>
<td>91/28</td>
<td>F. de Boer</td>
<td>A compositional proof system for dynamic process creation, p. 24.</td>
</tr>
</tbody>
</table>